struct

Motivazioni

Finora abbiamo utilizzato tipi *semplici*

- tipi numerici: float, int etc...
- array e puntatori

In molti casi è utile avere tipi più complessi...

Esempi

- punti su un piano cartesiano (coordinate)
- nodi di un albero (etichetta del nodo,figli)
- oggetti di una lista (elemento precedente/successivo)
- .....

Esempio: punti su un piano cartesiano

Un punto è una coppia di coordinate intere (x, y)

int x, int y

Rappresentazione mediante tipi elementari:

- Svantaggi:
  - Difficoltà di comprensione: variabili apparentemente scollegate
  - Difficile dichiarare, allocare o passare come parametri oggetti complessi
  - Molto complicata la gestione dei puntatori

struct (sintassi)

Possiamo creare un nuovo tipo point composto da due interi x e y

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

x e y sono campi della struct.

N.B. In una struct possono esserci campi di diversi tipi, anche altre struct.

Esempio

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


int main(){
  struct point p;  //dichiarazione

  p.x=10;                  //accesso ad un campo con “.”
  p.y=14;

  printf(“Posizione x:%d,y%d”,p.x,p.y);
}
```

struct e funzioni 1/2

Una funzione può restituire una struct:

```
struct point
{
   int x, y;
};

struct point createPoint(int x, int y)
{
   struct point p;
   p.x=x;
   p.y=y;
   return p;
}
```

N.B. Non c'è conflitto tra il campo **x** di **p** e la **x** passata come parametro alla funzione!!

struct e funzioni 2/2

Una struct può essere passata come parametro ad una funzione:

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

struct point sumPoints(struct point p1,struct point p2)
{
  p1.x=p1.x+p2.x;
  p1.y=p1.y+p2.y;
  return p1;      //Operiamo direttamente su p1 perché?
}
```

struct e funzioni 2/2

Una struct può essere passata come parametro ad una funzione:

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

struct point sumPoints(struct point p1,struct point p2)
{
  p1.x=p1.x+p2.x;
  p1.y=p1.y+p2.y;
  return p1;      //Operiamo direttamente su p1 perché?
}
```

Come nel caso dei tipi *semplici* i passaggi di parametro avvengono sempre per valore!

struct e puntatori

E' possibile dichiarare un puntatore ad una struct come si fa per una variabile ordinaria.

Dichiarazione:

```
struct point* p
```

Allocazione:

```
p= malloc(sizeof(struct point));
```

Assegnamento:

```
(*p).x=10 //Corretto
 p->x=10 //Equivalente alla precedente
*p.x=10  //SBAGLIATO
```

L'operatore “.” ha la precedenza rispetto a “*”

Passaggi per riferimento

In certi casi è meglio passare ad una funzione un puntatore ad una struct piuttosto che la struct stessa.

- Quando vogliamo modificare un campo della struct (necessario)
- Quando la struct è molto grande e la copia dei suoi valori può essere costosa. (consigliabile)

```
void swapPoint(struct point* p){
   int tmp=p->x;
   p->x=p->y;
   p->y=tmp;
}

int main(){
   struct point* p=malloc(sizeof(struct_point));
   p->x=5;
   p->y=2;
   swapPoint(p);
   printf("x:%d,y:%d",p->x,p->y);
}
```

typedef(una scorciatoia)

In C è possibile creare nuovi nomi di tipi di dato da utilizzare per maggiore leggibilità o compattezza.

```
typedef tipo nome_tipo;
```

Esempio:

```
typedef char* String;
```

```
int main(){
   String str=”Hello typedef”;
   printf(“%s”,str);
}
```

String può essere usata esattamente come char*

typedef con struct

Esempio di utilizzo di typedef con struct

```
typedef struct _point{
   int x, y;
} point;

int main(){
   point p;
   p.x=10;
   p.y=20;
}
```

N.B. Con lo strumento typedef non creiamo nuovi tipi ma associamo nuovi nomi a tipi già esistenti

struct ricorsive

E' possibile usare come campo di una struct un puntatore allo stesso tipo di struct.

```
struct point
{
   int x,y;
   struct point* p;   //Solo puntatori!
};
```

Utile per implentare strutture dati ricorsive come liste o alberi!