Strutture e tipi

Le strutture sono tipi di variabili in cui è possibile memorizzare dati eterogenei. Si possono considerare simili a vettori in cui ogni elemento può avere un tipo diverso, e in cui gli elementi sono identificati non da un numero ma da un nome.

Il linguaggio C fornisce un insieme di tipi di dati base, quali `int`, `char`, `float`, ecc. È però possibile definire nuovi tipi usando la primitiva `typedef`. I nuovi tipi di dato vengono in questo modo definiti a partire da quelli vecchi. Una volta definito un nuovo tipo, questo è del tutto equivalente ai tipi base del linguaggio.

Qui sotto vediamo per quale motivo le strutture e i tipi sono utili, e alcune loro applicazioni. Altre applicazioni, quali le liste e le strutture di dati, verranno spiegate nel dettaglio in un capitolo successivo.

Strutture

Strutture

Gli array, come si è visto, si possono vedere come un insieme di variabili, tutte dello stesso tipo. Dato un array, le variabili al suo interno sono identificate da un numero intero. Le strutture sono in un certo senso un concetto simile: una struttura è un insieme di variabili che possono anche avere tipi diversi. Al contrario dei vettori, per accedere agli elementi di una struttura si usa un nome invece che un numero.

	Array	Strutture
Tipi degli elementi	tutti uguali	qualsiasi tipo
Accesso al singolo elemento	attraverso un numero	attraverso un nome

Supponiamo di voler memorizzare in una variabile i dati di alcune persone. Per ogni persona, ci serve una stringa per il nome, una stringa per il cognome e un intero per l'età. Un semplice programma che usa i dati per una singola persona è il seguente nostruct.c

```
/*
  Dati di una persona, senza usare le strutture.
*/

#include<stdlib.h>

/* stampa i dati di una persona */

void stampapersona(char *nome, char *cognome, int eta) {
  printf("%s %s, eta' %d\n", nome, cognome, eta);
}


/* main */

int main() {
  char *nome;
  char *cognome;
  int eta;
```

```
  nome="Diego";
  cognome="Calvanese";
  eta=12;

  stampapersona(nome, cognome, eta);

  return 0;
}
```

Questo programma non presenta nessuna difficoltà, e funziona perfettamente: il nome e il congnome della persona vengono messi nelle variabili di tipo `char *` che abbiamo chiamato `nome` e `cognome`, rispettivamente. L'età viene invece memorizzata nella variabile intera `eta`. Questo programma presenta però un problema di leggibilità, e di modificabilità.

leggibilità
: non risulta subito chiaro che le tre variabili `nome`, `cognome`, ed `eta` sono tre attributi dello stesso oggetto;

modificabilità
: se vogliamo aggiungere la data di nascita, bisogna aggiungere una variabile, e poi modificare tutte le chiamate di funzione.

Il primo problema può causare errori se per esempio si chiama la funzione `stampapersona` commettendo l'errore di passare il nome di una persona, il cognome di un'altra e l'età di una altra persona ancora. Il secondo problema può rendere difficile modificare il codice, e questo può a sua volta portare a errori se il codice non viene modificato nel modo giusto.

Una soluzione a questi problemi è quella di definire una variabile di tipo persona, che contiene al suo interno tutti i dati di una persona, ossia nome, cognome ed età. Se non ci fosse l'età, questo si potrebbe fare con un vettore:

```
  char *p[2];
```

In questo modo, potremmo assumere che `p[0]` è il nome di una persona, mentre `p[1]` è il suo cognome. Si può quindi dire che la variabile `p` contiene sia il nome che il cognome di una singola persona. Il problema che resta è la variabile età, che è di tipo diverso dalle altre due: intero invece che puntatore a carattere.

Il C mette a disposizione la primitiva `struct` che consente di definire una variabile come un gruppo di variabili anche di tipo diverso. La definizione di una struttura avviene nel modo seguente:

```
struct Persona {
  char *nome;
  char *cognome;
  int eta;
};
```

Questa definizione ci dice che servono delle variabili composte, i cui elementi sono due puntatori a caratteri e un intero. In altre parole, dopo aver messo questo frammento di codice, possiamo definire delle variabili che sono composte da questi tre elementi. Per dichiarare che una variabile rappresenta una persona, ossia è composta da questi tre dati, usiamo una dichiarazione di questo tipo:

```
struct Persona x;
```

Si faccia attenzione: la prima dichiarazione dice che ci occorrono delle variabili composte da due stringhe e un intero. La seconda dice che la variabile `x` è una di queste variabili. Quindi, se si vogliono dichiarare tre variabili, ognuna delle quali rappresenta una persona, la prima dichiarazione si mette una sola, volta, e poi si dichiarano separatamente le tre variabili. Le seguenti dichiarazioni sono ‘‘quasi’’ equivalenti.

```
/* definizione della
struttura Persona */
struct Persona {
  char *nome;
  char *cognome;
  int eta;
};

/* dichiara tre
variabili strutture */
struct Persona x;
struct Persona a;
struct Persona w;
```

```
/* dichiarazione di x */
char *x_nome;
char *x_cognome;
int x_eta;

/* dichiarazione di y */
char *y_nome;
char *y_cognome;
int y_eta;

/* dichiarazione di z */
char *z_nome;
char *z_cognome;
int z_eta;
```

Si noti che in C non è in effetti possibile dichiarare delle variabili che contengono il punto nel nome. Questo è possibile solo dichiarando nelle strutture, in cui il punto separa il nome della variabile dal nome dell’elemento della struttura. Le due dichiarazioni di sopra sono quasi equivalenti nel senso che dove quella di destra dichiara una variabile stringa `x_nome`, quella di destra implicitamente definisce `x.nome` come una variabile stringa, ecc.

Per poter usare (leggere o scrivere) le tre componenti di una di queste variabili, si usa il nome della variabile seguita da un punto e dal nome che abbiamo scritto dopo il tipo nella definizione di struttura. In altre parole, dopo le dichiarazioni di sopra, sono definite le due variabili `x.nome` e `x.cognome` di tipo stringa, e la variabile `x.eta` di tipo intero.

La dichiarazione della struttura `Persona` contiene al suo interno tre righe, ognuna delle quali ha esattamente la forma di una dichiarazione di variabile. Tuttavia, nessuna di queste dichiarazioni definisce una variabile. Al contrario, servono a indicare che, ogni volta che si definisce una nuova variabile struttura con quel nome, essa è costituita da tre parti, e fornisce il tipo e il nome di ciasuna delle tre parti. Si noti la differenza fra dichiarazione di struttura, di variabile, e di variabile struttura:

```
               /* definizione della struttura Persona */
struct Persona {
  char *nome;          /* indica che ogni variabile struttura ha
                       una parte di tipo char *, e si puo' accedere
                       a questa parte usando .nome */
  char *cognome;       /* ogni variabile struttura contiene un char *
                       a cui si puo' accedere con .cognome */
  int eta;             /* ogni variabile struttura contiene un intero
```

```
                        a cui si puo' accedere con .eta */
};

int max_eta;            /* definisce una variabile di tipo int */

struct Persona x;       /* definisce la variabile struttura x, ed
                        e' equivalente a definire tre variabili x.nome,
                        x.cognome, e x.eta (le prime due di tipo char *,
                        la terza di tipo int */
```

In altre parole, quello che sta dentro la definizione di struttura non è la dichiarazione di una variabile, ma indica che quando si dichiara una variabile struttura si dichiarano automaticamente le tre variabili che hanno il nome della variabile seguito da `.nome`, da `.cognome` e da `.eta`, di tipi `char *` e `int`.

È possibile passare una intera struttura a una funzione: basta che la definizione di struttura preceda la definizione della funzione. Nella funzione la definizione del parametro è uguale alla dichiarazione di una variabile, come sempre. Nell'esempio, possiamo fare in modo che la funzione `stampapersona` prenda come parametro una variabile struttura invece delle tre variabili separate. Questo si fa semplicemente mettendo fra parentesi tonde una dichiarazione di variabile struttura:

```
void stampapersona(struct Persona p) { ...
```

All'interno della procedura risulta definita una variabile struttura `p`, i cui campi coincidono con quelli della variabile con cui la procedura è stata chiamata. In altre parole, le variabili struttura si comportano come tutte le altre variabili anche rispetto alle chiamate di procedura: si può passare una variabile struttura a una funzione, e la cosa avviene come per tutte la altre variabili.

Il programma construct.c è una versione modificata del precedente, in cui si usa una struttura invece di tre variabili.

```
/*
  Definizione di una struttura che contiene tutti
i dati di una persona.
*/

#include<stdlib.h>

/* definizione della struttura */

struct Persona {
  char *nome;
  char *cognome;
  int eta;
};


/* stampa i dati di una persona */

void stampapersona(struct Persona p) {
  printf("%s %s, eta' %d\n", p.nome, p.cognome, p.eta);
}


/* main */

int main() {
  struct Persona x;
```

```
  x.nome="Diego";
  x.cognome="Calvanese";
  x.eta=12;

  stampapersona(x);

  return 0;
}
```

Per quello che riguarda la modificabilità, notiamo che aggiungere una nuova componente a una struttura comporta solo la modifica della dichiarazione della struttura. Per esempio, per aggiungere la data di nascita, basta questo:

```
struct Persona {
  char *nome;
  char *cognome;
  int eta;
  int giorno_nascita;
  int mese_nascita;
  int anno_nascita;
};
```

e automaticamente tutte le variabili struttura dichiarate nel programma avranno una componente di tipo `int` e nome `giorno_nascita`, ecc.

Terminologia: le componenti di una struttura si chiamano *campi* della struttura. Quindi, il campo di una struttura è una parte delle variabili struttura. Nell'esempio di sopra, la struttura `Persona` ha tre campi: `nome`, `cognome` e `eta`.

Strutture per rappresentare array dinamici

Strutture per rappresentare array dinamici

Gli array dinamici presentano una tipica situazione in cui servono più variabili per rappresentare lo stesso oggetto. Infatti, un array dinamico viene rappresentato come un puntatore che indica la zona di memoria in cui i dati sono memorizzati, e un numero che dice la grandezza dell'array, ossia il numero di elementi allocati.

Dal punto di vista della comodità di scrittura del codice, può risultare comodo utilizzare una sola variabile che contenga questi due dati. Usiamo quindi una struttura a due campi (i campi di una struttura sono la parti componenti delle variabili di tipo struttura). Il primo campo è il vettore stesso, ossia un puntatore al tipo che costituisce il vettore (per esempio, un intero); il secondo campo è un numero intero, che rappresenta il numero di elementi che sono stati allocati per il vettore. La dichiarazione della struttura è quindi la seguente:

```
struct ArrayDinamico {
  int *punt;
  int dim;
};
```

In questo modo, ogni volta che si dichiara una variabile struttura, questa è automaticamente composta da un puntatore a interi (quindi un vettore di interi) e da un intero.

Per dichiarare una variabile struttura, si usa una dichiarazione di tipo come segue:

```
  struct ArrayDinamico a;
```

Come si è detto prima, questa dichiarazione crea una variabile `a.punt` di tipo puntatore a intero, e una variabile `a.dim` di tipo intero.

Si noti che il valore di queste due variabili è completamente indipendente: possiamo anche per esempio allocare 100 elementi per il vettore e scrivere il numero 200 nella variabile intera. Questo è ammesso dal linguaggio, ma è un errore di programmazione, dal momento che la variabile viene usata per rappresentare appunto il numero di interi allocati per il vettore. Tutto questo vuole dire che è il programmatore che deve fare in modo che il campo `dim` contenga sempre il numero di elementi allocati per il vettore, e non è il calcolatore che automaticamente aggiorna questo valore.

Questo significa che, quando si alloca (`malloc`) oppure si rialloca (`realloc`) il vettore, occorre aggiornare anche il valore di `a.dim` che rappresenta il numero di elementi contenuti nel vettore.

Il seguente programma arraystruct.c definisce una variabile struttura `a`. Il vettore viene allocato, e vengono scritti alcuni valori al suo interno. Viene anche definita una funzione che stampa tutti gli elementi del vettore. Si noti che questa funzione usa sia il puntatore che la dimensione, ma viene chiamata passando semplicemente la variabile struttura.

```
/*
  Array dinamico definito come una struttura.
*/

#include<stdlib.h>

struct ArrayDinamico {
  int *punt;
  int dim;
};

void StampaVettore(struct ArrayDinamico x) {
  int i;

  for(i=0; i<=x.dim-1; i++)
    printf("%d ", x.punt[i]);

  printf("\n");
}

int main() {
  struct ArrayDinamico a;
  int i;

  a.dim=10;
  a.punt=malloc(10*sizeof(int));

  for(i=0; i<=5; i++)
    a.punt[i]=i*i;

  StampaVettore(a);

  return 0;
}
```

Vettori parzialmente utilizzati

Nel precedente esercizio si è visto come sia possibile usare le strutture per memorizzare può dati riguardanti lo stesso oggetto. In particolare, usiamo una struttura per memorizzare un vettore dinamico e la sua dimensione. È però facile rendersi conto che questi due dati non sono sufficienti nel caso in cui solo una parte del vettore contiene dei valori significativi. In questo caso, serve un terzo campo che indica il numero di elementi significativi del vettore.

Il programma parziale.c è una modifica del precedente, in cui la struttura contiene un terzo campo di tipo intero, di nome `n`, che contiene il numero di elementi significativi del vettore.

Anche in questo caso (come del resto nel precedente) è compito del programmatore garantire che il valore memorizzato in `a.n` sia effettivamente il numero di elementi significativi del vettore. Questo si può realizzare mettendo inizialmente a 0 questa variabile; ogni volta che viene immesso un nuovo valore in `a.punt[i]`, si va a vedere se sto creando un nuovo valore significativo oppure no. Questo non è difficile da controllare: infatti, se il vettore ha `a.n` elementi significativi, allora questi hanno indici `0`, `1`, ... , `a.n-1`. Quindi, se `i` è maggiore di `a.n-1`, allora sto mettendo un valore in una posizione che precedentemente non era significativa del vettore. Le nuove posizioni significative del vettore sono ora quelle di indici `0`, `1`, ... , `i`. Da questo segue che il nuovo valore di `a.n` deve essere tale che `a.n-1` è uguale a `a.i`, per cui occorre assegnare `a.n=i+1`. Questo si può riassumere come segue:

```
elementi significativi prima: 0 1 2 3 ... a.n-1
elementi significativi dopo:  0 1 2 3 ... a.n-1 ... i
```

Questo significa che dopo l'assegnamento ho `i+1` elementi significativi, e questo deve essere il nuovo valore di `a.n`.

Il programma completo è riportato qui sotto. È una versione modificata del precedente, in cui oltre a tenere il valore di `a.dim` pari al numero di elementi allocati per il vettore, metto in `a.n` il numero di elementi significativi.

```
/*
  Array dinamico definito come una struttura.
*/

#include<stdlib.h>

struct ArrayDinamico {
  int *punt;
  int dim;
  int n;
};

void StampaVettore(struct ArrayDinamico x) {
  int i;

  for(i=0; i<=x.n-1; i++)
    printf("%d ", x.punt[i]);

  printf("\n");
}
```

```
int main() {
  struct ArrayDinamico a;
  int i;

  a.dim=10;
  a.punt=malloc(10*sizeof(int));
  a.n=0;

  for(i=0; i<=5; i++) {
    a.punt[i]=i*i;
    if(a.n<i+1)
      a.n=i+1;
  }

  StampaVettore(a);

  return 0;
}
```

Si noti che la dichiarazione di variabile, e la intestazione della procedura, sono rimaste inalterate. Uno dei vantaggi di usare una struttura al posto delle variabili separate è che in questo modo l'aggiunta di nuovi campi comporta del codice solo nei punti in cui questi nuovi campi vengono usati.

Lettura array da file

Lettura array da file

Vediamo ora un esercizio sugli array dinamici. Si tratta semplicemente di riscrivere l'esercizio di lettura di array da file, in cui l'array viene rappresentato con una struttura invece che con tre variabili separate che indicano il vettore stesso, la sua dimensione, e il numero di elementi usati.

Il programma inizia con una allocazione iniziale dell'array: vengono allocati 100 elementi, e quindi la variabile che indica la dimensione viene aggiornata. In questo momento l'array contiene 100 elementi, ma nessuno è stato ancora letto: possiamo quindi dire che nessuno di questi cento elementi è usato per rappresentare un valore significativo.

Il programma prosegue controllando se è stato lanciato con il numero giusto di argomenti, e in caso affermativo apre il file il cui nome è stato passato come argomento.

Vediamo ora più in dettaglio il ciclo di lettura:

```
                        /* ciclo di lettura */
  i=0;
  while(1) {
    res=fscanf(fd, "%d", &a.punt[i]);
    if( res!=1 )
      break;

    if(a.n<i+1)
      a.n=i+1;

    i++;

    if(i>a.dim-1) {
```

```
      a.dim+=100;
      a.punt=realloc(a.punt, a.dim*sizeof(int));
    }
  }
```

Usiamo una variabile `i` per scandire l'array. Dal momento che il primo elemento letto è quello di indice zero, questa variabile parte da zero.

Si inizia poi il ciclo: si legge un elemento dell'array, e se la cosa non riesce si esce dal ciclo. Se l'elemento `a.punt[i]` è stato letto con successo, allora l'elemento del vettore di indice `i` contiene un valore significativo, quindi il vettore ora ha `i+1` elementi significativi (di indice da `0` a `i`). Dal momento che il valore `a.n` deve rappresentare il numero di elementi significativi dell'array, va aggiornato se è inferiore a `i+1`.

A questo punto, si aumenta il valore dell'indice `i` per prepararsi alla lettura del prossimo elemento. L'elemento che verrà letto al prossimo ciclo à `a.punt[i]`. Non è detto però che questo elemento sia stato già allocato. Quindi, occorre controllare se il vettore ha abbastanza elementi allocati. Dal momento che nel vettore ci sono `a.dim` elementi allocati, allora questi hanno indici `0`, `1`, ..., `a.dim-1`. Se `i` è al di fuori di questo intervallo, ossia `i>a.dim-1`, allora va fatta una nuova allocazione di memoria.

A volte è possibile che ci sia confusione su quando mettere il `-1` e quando no. La regola generale da seguire è che i primi `x` elementi di un vettore hanno indici `0`, `1`, ..., `x-1`. Se quindi voglio sapere se l'indice `i` si trova fra i primi `x` elementi del vettore, devo controllare se si trova in quell'intervallo.

Il programma completo leggiarray.c è riportato qui sotto.

```
/*
  Legge da file un array dinamico,
che viene rappresentato come struttura.
*/

#include<stdlib.h>
#include<stdio.h>


/* la struttura che contiene il vettore dinamico,
la sua dimensione, e il numero di elementi effettivamente
usati */

struct ArrayDinamico {
  int *punt;
  int dim;
  int n;
};


/* stampa un vettore dinamico su schermo */

void StampaVettore(struct ArrayDinamico x) {
  int i;

  for(i=0; i<=x.n-1; i++)
    printf("%d ", x.punt[i]);

  printf("\n");
}
```

```
/* main */

int main(int argn, char *argv[]) {
  FILE *fd;
  int res;

  struct ArrayDinamico a;
  int i;


                        /* inizializza l'array */
  a.dim=100;
  a.punt=malloc(a.dim*sizeof(int));
  a.n=0;                /* non ho letto nessun elemento finora */


                        /* controllo argomenti */
  if( argn-1 < 1 ) {
    printf("Manca il nome del file che contiene il vettore\n");
    exit(1);
  }


                        /* apre il file */
  fd=fopen(argv[1], "r");
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


                        /* ciclo di lettura */
  i=0;
  while(1) {
    res=fscanf(fd, "%d", &a.punt[i]);
    if( res!=1 )
      break;

    if(a.n<i+1)
      a.n=i+1;

    i++;

    if(i>a.dim-1) {
      a.dim+=100;
      a.punt=realloc(a.punt, a.dim*sizeof(int));
    }
  }


                        /* chiude il file */
  fclose(fd);


                        /* stampa il vettore */
  StampaVettore(a);

  return 0;
}
```

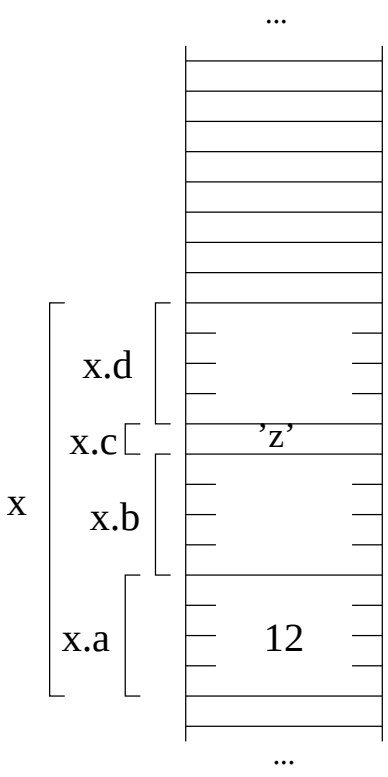
Indirizzi e puntatori a strutture

Indirizzi e puntatori a strutture

Si può determinare sia l'indirizzo della struttura che l'indirizzo di ogni suo elemento. L'operatore che si usa è ancora l'ampersand &. Il seguente programma indstruct.c trova e stampa gli indirizzi di una variabile struttura e dei suoi elementi.

```
/*
  Indirizzi di una struttura.
*/

#include<stdlib.h>
#include<stdio.h>

/* definisce una struttura */

struct Esempio {
  int a;
  float b;
  char c;
  char *d;
};


int main() {
  struct Esempio x;

  printf("Indirizzo di x: %X\n", &x);
  printf("Indirizzo di x.a: %X\n", &x.a);
  printf("Indirizzo di x.b: %X\n", &x.b);
  printf("Indirizzo di x.c: %X\n", &x.c);
  printf("Indirizzo di x.d: %X\n", &x.d);

  return 0;
}
```

La figura accanto dà una rappresentazione grafica dello spazio occupato.



Come per i tipi scalari, si possono definire dei puntatori a struttura, in cui possono venire memorizzati indirizzi di strutture. Nell'esempio di sopra, se si definisce una variabile con:

```
struct Esempio *p;            /* puntatore a struttura */
```

Questo definisce `p` come una variabile il cui contenuto è un indirizzo di memoria. In particolare, è l'indirizzo in cui inizia la zona di memoria in cui è memorizzata una struttura `Esempio`.

Dal momento che i puntatori a strutture sono variabili che contengono indirizzi di strutture, è lecito fare un assegnamento `p=&x`, che mette nella variabile `p` l'indirizzo iniziale di `x`.

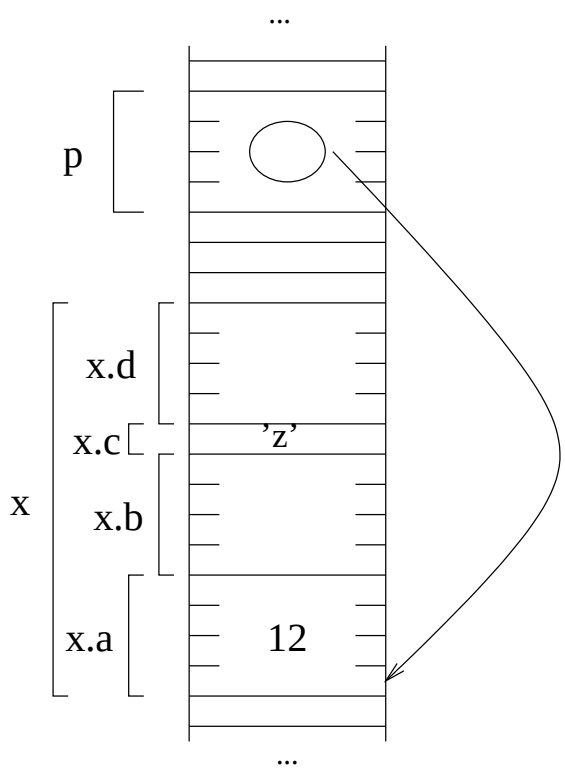
Se `p` è un puntatore a struttura, allora `*p` è ovviamente la struttura puntata. In altre parole, se si fa `p=&x`, allora scrivere `x` oppure `*p` è esattamente la stessa cosa. Quindi, per esempio si può passare a una funzione che ha una struttura `Esempio` come parametro sia `x` che `*p`.

Dal momento che `*p` è una struttura, allora si può anche accedere ai suoi campi. Si faccia però attenzione alle regole di precedenza: si deve scrivere `(*p).a` per accedere al campo `a` della struttura, mentre scrivere `*p.a` produce un errore. Nel caso dell'esempio, `(*p).a` è equivalente a `x.a`.

Questa espressione si può abbreviare usando l'operatore freccia: al posto di `(*p).a` si può scrivere `p->a`, che migliora di molto la leggibilità. In generale, la espressione `puntatore->campo` è equivalente a `(*puntatore).campo`, ossia identifica il campo della struttura puntata. Questa notazione verrà usata molto nell'ambito delle strutture collegate.

Il seguente programma puntstruct.c mostra un uso dei puntatori a strutture.

```
/*
  Indirizzi di una struttura.
*/

#include<stdlib.h>
#include<stdio.h>

/* definisce una struttura */

struct Esempio {
  int a;
  float b;
  char c;
  char *d;
};


int main() {
  struct Esempio x;
  struct Esempio *p;            /* puntatore a struttura */

  p=&x;                         /* prende l'indirizzo della struttura */
                                /* da ora in poi, *p e x sono la stessa cosa */

  (*p).a=12;
  printf("Valore di x.a: %d\n", x.a);

  x.c='z';
  printf("Valore di x.c: %c\n", (*p).c);

                                /* -> e' equivalente a "(* )." */
  printf("Valore di x.c: %c\n", p->c);

  return 0;
}
```



La figura qui accanto mostra lo stato della memoria alla fine della esecuzione del programma: la struttura è la stessa del programma precedente, e in più abbiamo una variabile `p` che contiene l'indirizzo del punto iniziale della zona di memoria in cui si trova la struttura `x`.

Array di strutture

Array di strutture

Si possono definire array di un qualsiasi tipo numerico, come `int`, `float`, ecc. È però anche possibile definire array di puntatori, e array di strutture. Le regole sono esattamente identiche: un array è una sequenza di variabili di uno stesso tipo, e ogni variabile è identificata da un indice numerico.

Se quindi nel programma il tipo struttura `struct Esempio` è definito, allora si può definire una variabile di tipo array di strutture, nel seguente modo:

```
  struct Esempio vett[100];
```

Questo definisce cento variabili di tipo struttura, ognuna identificata da un indice numerico. Quindi, `vett[0]` è una variabile struttura, come anche `vett[1]`, ecc.

Il programma seguente array.c risolve usando un array di strutture il problema seguente: dato un file che contiene, per ogni riga, una coppia composta da un intero e una stringa, che rappresentano la dimensione di un file e il suo nome (senza estensione), trovare il nome del file più grande. Per esempio, nel file files.txt il file più grande è `puntind`, che ha dimensione 12.

Questo esercizio si può risolvere come segue: per ogni file memorizziamo la sua dimensione e il suo nome in una struttura. Quindi, per memorizzare un singolo file, usiamo una struttura con due campi, che sono un intero e una stringa. La definizioe della struttura è quindi la seguente:

```
struct DimFile {
  int dimensione;
  char nome[100];
};
```

Procediamo in questo modo: facciamo una scansione del file, memorizzando i dati di ogni file in una struttura di tipo `struct DimFile`. Dal momento che abbiamo bisogno di un certo numero di variabili di questo tipo, usiamo un vettore di strutture:

```
  struct DimFile files[100];
```

Da questo momento in poi, `files[0]`, `files[1]`, ecc sono variabili di tipo struttura. Quindi, quando si legge la prima riga, possiamo memorizzare la dimensione nella variabile intera `files[0].dimensione` e la il suo nome nella variabile stringa `files[0].nome`.

Nel dettaglio, il programma risolutivo che usiamo legge tutto il file e lo memorizza nel vettore di strutture. Per trovare il massimo, fa una scansione del vettore, memorizzando nella variabile intera `max` *l'indice* del vettore in cui è memorizzato il file più grande incontrato fino ad ora (questa variabile parte a 0, e viene aggiornata ogni volta che si trova un elemento di dimensione maggiore del massimo). Si noti che, essendo `max` l'indice della struttura che contiene l'elemento più grande del vettore, allora il nome del file più grande è quello memorizzato in `files[max].nome`, e la sua dimensione si trova in `files[max].dimensione`.

```
/*
  Trova il nome del file piu' grande.
Esempio di uso di array di strutture.
*/

#include<stdlib.h>
#include<stdio.h>

/* una struttura che contiene la dimensione di
un file e il suo nome */

struct DimFile {
  int dimensione;
  char nome[100];
};

int main() {
  FILE *fd;
  int res;

  struct DimFile files[100];
  int n;

  int i, max;


               /* apre il file */
  fd=fopen("files.txt", "r");
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


               /* legge il file */
  n=0;
  while(1) {
                       /* legge la dimensione */
    res=fscanf(fd, "%d", &files[n].dimensione);
    if(res!=1)
      break;
```

```
                        /* legge il nome */
    res=fscanf(fd, "%s", &files[n].nome);
    if(res!=1) {
      printf("Errore in lettura del nome di file\n");
      exit(1);
    }

    n++;
  }



                /* trova il massimo */
  max=0;
  for(i=0; i<=n-1; i++)
    if( files[max].dimensione<files[i].dimensione )
      max=i;


                /* stampa il massimo */
  printf("Il file piu' lungo e' %s. Ha dimensione %d\n",
         files[max].nome, files[max].dimensione);


  return 0;
}
```


Definizione di nuovi tipi

Il C mette a disposizione un insieme di tipi di dati piuttosto ristretto: `int`, `char`, `float`, ecc. È però anche possibile definire dei nuovi tipi. Iniziamo dal caso più semplice, ossia quello in cui vogliamo semplicemente realizzare un sinonimo di un nome di tipo che non ci piace. Per esempio, possiamo pensare che il tipo `int` abbia un nome poco comprensibile, e che al suo posto vorremmo usare un nuovo nome di tipo `intero`. Questo si realizza cosí:

```
typedef int intero;
```

Da questo punto in poi, si può usare `int` oppure `intero` indifferentemente. In altre parole, questa definizione dice semplicemente che `intero` è un sinonimo di `int`. Per esempio, il seguente programma tipoint.c, dopo la definizione di tipo dichiara una variabile di tipo `intero`, che come si è detto è ora equivalente a `int`.

```
/*
  Definisce un nuovo tipo.
*/

typedef int intero;

int main() {
  intero x;

  x=10;
```

```
  printf("x vale %d\n", x);

  return 0;
}
```

Esistono casi in cui la definizione di un nuovo tipo non è cosí semplice. Questi casi verranno analizzati nella prossima pagina. Si noti anche che il nuovo nome di tipo `intero` è equivalente al vecchio nome `int` in tutto e per tutto: per esempio si può usare per dichiarare quale tipo di parametri ha una funzione, e si può usare la funzione `sizeof` per determinare lo spazio allocato per le variabili di questo tipo.

Altri esempi di tipi

Altri esempi di tipi

La definizione di un nuovo tipo equivalente a un vecchio tipo numerico è il caso più semplice. Ora vediamo dei casi più complessi. Iniziamo dal caso in cui definiamo il tipo vettore. La definizione di tipo risulta in questo caso più complessa:

```
typedef int TipoVettore[100];
```

Questa definizione non è facile da interpretare. Assomiglia alla dichiarazione di una variabile vettore di tipo intero, con 100 elementi. In effetti, fa qualcosa di simile: definisce un nuovo nome di tipo `TipoVettore`, in modo tale che le variabili di questo tipo sono vettori di interi di 100 elementi.

In altre parole, dopo la definizione di sopra, se si fa una dichiarazione di tipo:

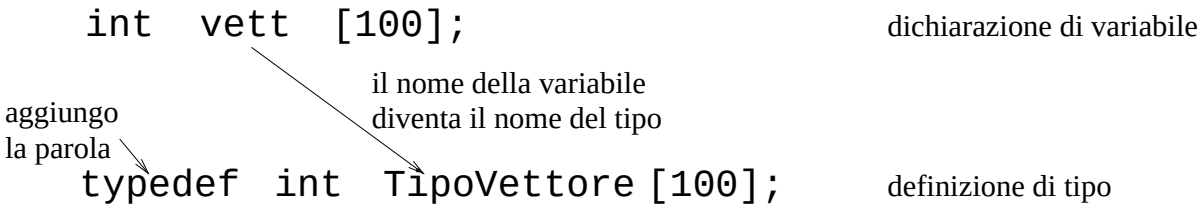
```
TipoVettore x, y;
```

Allora `x` e `y` sono definite come array di interi con cento elementi.

Questa regola generale di definizione di tipo si può riassumere come segue:

> *una definizione di tipo è simile alla definizione di una variabile, ma ha la parola chiave `typedef` davanti;*
>
> *quello che sarebbe il nome della variabile diventa il nome del tipo;*
>
> *le variabili definite di questo tipo sono del tipo che si otterebbe mettendo il nome della variabile al posto del tipo e togliendo `typedef`.*

Seguendo queste regole, se è possibile dichiarare una variabile di un certo tipo, allora è anche possibile dichiarare un tipo. Dato che possiamo definire variabili di tipo vettori di interi di cento elementi, allora possiamo anche definire il tipo ‘‘vettore di interi di cento elementi’’, e questo si ottiene prendendo una dichiarazione di variabile, sostituendo al nome di variabile il nome che abbiamo deciso di usare per il nuovo tipo e mettendo `typedef` davanti.



Da questo momento `TipoVettore` rappresenta un nuovo nome di tipo. Quindi, possiamo definire delle variabili di questo tipo usando una semplice dichiarazione:

```
TipoVettore x, y;
```

Per capire come esattamente è definita la variabile `x`, bisogna considerare la dichiarazione del tipo, cancellare `typedef` e mettere `x` al posto del nome del tipo. Quello che si ottiene è:

```
typedef int TipoVettore[100];
        int           x[100];
```

Ossia `x` è un vettore di cento interi. Si noti che questo meccanismo è *implicito* nella dichiarazione della variabile `x`. In altre parole, `TipoVettore x;` è equivalente a `int x[100];`, per cui non è necessario (anzi, è un errore) scrivere esplicitamente la dichiarazione `int x[100];`. La variabile `y` è ovviamente definita nello stesso modo, per cui è anche essa un vettore di cento interi.

Il seguente programma tipovettore.c definisce il tipo vettore, dichiara una variabile di questo tipo, e la usa.

```
/*
  Definisce un tipo per un vettore.
*/

typedef int TipoVettore[100];

int main() {
  TipoVettore x;
  int i;

  for(i=0; i<=10; i++)
    x[i]=i*i;

  for(i=0; i<=10; i++)
    printf("%d ", x[i]);

  printf("\n");

  return 0;
}
```

Riassumendo: se ho una variabile fatta in un certo modo, posso definire un tipo corrispondente. Si possono poi definire variabili di quel tipo, che risultano definite in modo analogo alla variabile di partenza.

Questo meccanismo di definizione di nuovi tipi, come si è detto, è applicabile a tutti i tipi di variabili. Per esempio, dato che è possibile definire una variabile struttura, possiamo definire un tipo. Una variabile struttura si dichiara con:

```
struct NomeStruttura variabile;
```

Il tipo corrispondente si ottiene mettendo `typedef` davanti, e mettendo il nome che si vuole dare al tipo al posto del nome della variabile. Quello che si ottiene è:

```
typedef struct NomeStruttura TipoNomeStruttura;
```

Da questo momento in poi, `TipoNomeStruttura` è un nome di tipo. Possiamo quindi definire delle variabili di questo tipo. Queste variabili sono costruite nello stesso modo della variabile da cui siamo partiti, quindi sono delle strutture. Per essere più precisi, una dichiarazione come `TipoNomeStruttura x;`, applicando la regola, è equivalente a `struct NomeStruttura x;`.

Il programma tipostruttura.c definisce un tipo per delle strutture usate per memorizzare i dati di un lotto di articoli, e che è in particolare composta dal peso degli articoli e dalla loro quantità. Viene dichiarata una variabile di tipo struttura, e usata.

```
/*
  Definisce un tipo per una struttura.
*/

struct Lotto {
  float peso;
  int quantita;
};

typedef struct Lotto TipoLotto;

int main () {
  TipoLotto x;

  x.peso=1.2;
  x.quantita=10;

  printf("Peso=%f quantita'=%d\n", x.peso, x.quantita);

  return 0;
}
```

Questo ultimo esempio riguarda i puntatori. Dato che possiamo definire delle variabili che sono puntatori, possiamo definire un tipo. Consideriamo il caso delle di una struttura definita per rappresentare i dati di un lotto di merce. Le sue componenti sono: il peso degli articoli, e poi la quantità di articoli.

```
struct Lotto {
  float peso;
  int quantita;
};
```

Possiamo definire una variabile struttura, come segue:

```
struct Lotto x;
```

Possiamo quindi definire un tipo struttura usando la solita regola:

```
typedef struct Lotto TipoLotto;
```

Dato che `TipoLotto` è un tipo equivalente a una struttura, possiamo definire variabili come strutture dichiarandole come `TipoLotto x`. Possiamo però anche dichiarare puntatori a struttura come:

```
TipoLotto *p;
```

Ora possiamo applicare ancora la regola, e definire un tipo per i puntatori a struttura, mettendo `typedef`, e poi sostituendo a `p` il nome che vogliamo dare al tipo.

```
typedef TipoLotto * TipoPuntLotto;
```

Queste due definizioni di tipo, insieme, definiscono il tipo `TipoPuntLotto` come il tipo delle variabili puntatore alla struttura `Lotto`. In altre parole, quando si definisce una variabile come `TipoPuntLotto p` questo è un puntatore a una struttura `Lotto`.

Il seguente programma tipopuntatore.c è un esempio di uso di queste definizioni.

```
/*
  Definisce un tipo per un puntatore a strutture.
*/

struct Lotto {
  float peso;
  int quantita;
};

typedef struct Lotto TipoLotto;
typedef TipoLotto * TipoPuntLotto;

int main () {
  TipoLotto x;
  TipoPuntLotto p;

  p=&x;

  p->peso=1.2;
  p->quantita=10;

  printf("Peso=%f quantita'=%d\n", x.peso, x.quantita);

  return 0;
}
```

Un modo alternativo di definire questo tipo si ottiene partendo dalla considerazione che una variabile puntatore a struttura si dichiara come:

```
struct Lotto *p;
```

Per cui si può definire un tipo cosí:

```
typedef struct Lotto *TipoPuntLotto;
```

Lotti di merce

Lotti di merce

lotti.c

```
/*
  Legge un file di lotti, e ne memorizza ognuno
in una struttura.
*/

#include<stdlib.h>
#include<stdio.h>

struct Lotto {
  float peso;
  int quantita;
};

typedef struct Lotto TipoLotto;

int main() {
  FILE *fd;
  int res;

  TipoLotto l;


                        /* apre il file */
  fd=fopen("lotti.txt", "r");
  if(fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }


                        /* ciclo di lettura del file */
  while(1) {
    res=fscanf(fd, "%f %d", &l.peso, &l.quantita);
    if( res!=2 )
      break;

    printf("Peso=%f Quantita'=%d\n", l.peso, l.quantita);
  }


                        /* chiude il file */
  fclose(fd);

  return 0;
}
```

---

lottivettore.c

```
/*
  Legge un file di lotti, memorizzandoli
in un vettore di strutture.
*/

#include<stdlib.h>
#include<stdio.h>
```

```
struct Lotto {
  float peso;
  int quantita;
};

typedef struct Lotto TipoLotto;


int main() {
  FILE *fd;
  int res;

  TipoLotto l;
  TipoLotto vett[100];
  int n;
  int i;


                        /* apre il file */
  fd=fopen("lotti.txt", "r");
  if(fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }


                        /* ciclo di lettura del file */
  n=0;
  while(1) {
    res=fscanf(fd, "%f %d", &l.peso, &l.quantita);
    if( res!=2 )
      break;

    vett[n]=l;
    n++;
  }


                        /* ciclo di stampa */
  for(i=0; i<=n-1; i++)
    printf("Peso=%f Quantita'=%d\n", vett[i].peso, vett[i].quantita);


                        /* chiude il file */
  fclose(fd);

  return 0;
}
```

---

lottiordine.c

```
/*
  Legge un file di lotti, stampa prima
quelli di peso 0.25, poi quelli di peso
0.5 e poi quelli di peso 1.
*/

#include<stdlib.h>
#include<stdio.h>
```

```
struct Lotto {
  float peso;
  int quantita;
};

typedef struct Lotto TipoLotto;


int main() {
  FILE *fd;
  int res;

  TipoLotto l;
  TipoLotto vett[100];
  int n;
  int i;


                        /* apre il file */
  fd=fopen("lotti.txt", "r");
  if(fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }


                        /* ciclo di lettura del file */
  n=0;
  while(1) {
    res=fscanf(fd, "%f %d", &l.peso, &l.quantita);
    if( res!=2 )
      break;

    vett[n]=l;
    n++;
  }


                        /* stampa i lotti di peso 0.25 */
  printf("Lotti di peso 0.25\n");
  for(i=0; i<=n-1; i++)
    if(vett[i].peso==0.25)
      printf("Quantita'=%d\n", vett[i].quantita);


                        /* stampa i lotti di peso 0.5 */
  printf("Lotti di peso 0.5\n");
  for(i=0; i<=n-1; i++)
    if(vett[i].peso==0.5)
      printf("Quantita'=%d\n", vett[i].quantita);


                        /* stampa i lotti di peso 1 */
  printf("Lotti di peso 1\n");
  for(i=0; i<=n-1; i++)
    if(vett[i].peso==1)
      printf("Quantita'=%d\n", vett[i].quantita);


                        /* chiude il file */
```

```
  fclose(fd);

  return 0;
}
```


Punti in un piano cartesiano

Supponiamo di voler rappresentare dei punti di un piano cartesiano. Come è noto, ogni punto si può rappresentare come una coppia di numeri reali, che danno le due coordinate che individuano un punto. Per ogni punto ci servono quindi due variabili. Dal momento che si tratta di due dati che riguardano lo stesso oggetto, è ragionevole usare una struttura:

```
struct Punto {
  float x;
  float y;
};

typedef struct Punto TipoPunto;
```

Queste due definizioni fanno sí che si possa dichiarare una variabile di tipo `TipoPunto`. Queta variabile risulta composta da due reali, che usiamo per memorizzare le coordinate `x` e `y` del punto. Il vantaggio è che è ora possibile rappresentare un punto come una singola variabile di tipo struttura, invece di avere due variabili indipendenti reali.

Per dichiarare una variabile come un punto, usiamo la solita sintassi della dichiarazione di variabile, mettendo `TipoPunto` come tipo, ossia:

```
TipoPunto e;
```

Questa dicharazione dice che la variabile `e` rappresenta un punto. In particolare, le coordinate sono memorizzate nei due campi `e.x` ed `e.y`. Quindi, per memorizzare il punto che ha coordinate `1.2` e `-2.2` in questa variabile, l'istruzione da usare è:

```
e.x=1.2;
e.y=-2.2;
```

Vediamo ora un piccolo esempio. Supponiamo di volere una funzione che calcola la distanza fra due punti. Questa funzione ritorna un numero reale. Per quello che riguarda i suoi parametri, abbiamo due alternative: o passiamo quattro parametri (due coordinate per ogni punto), oppure passiamo solo due parametri, ognuno dei quali è la struttura che contiene uno dei due punti.

La prima soluzione è:

```
float distanza(int x1, int y1, int x2, int y2) {
  float d;

  d=sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));

  return d;
}
```

La seconda soluzione consiste nel passare due strutture. Le coordinate dei due punti si ottengono semplicemente prendendo i due campi di ognuna di queste strutture.

```
float distanza(TipoPunto a, TipoPunto b) {
  float d;

  d = sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));

  return d;
}
```

Il vantaggio di questa seconda soluzione è evidente: ogni volta che si deve calcolare la distanza fra due punti, basta passare alla funzione i due punti, ossia le due variabili struttura che rappresentano i punti. Questo semplifica il codice e riduce la probabilità di commettere errori.

Il seguente programma punti.c è un esempio di quello che si è detto sopra: si definisce la struttura per rappresentare i punti, si definisce una funzione che calcola la distanza, e la si usa per calcolare la distanza fra due punti.

```
/*
  Definizione del tipo punto,
funzione che calcola la distanza.
*/

#include<stdlib.h>
#include<math.h>


/* definizione di struttura e tipo */

struct Punto {
  float x;
  float y;
};

typedef struct Punto TipoPunto;


/* distanza fra due punti */

float distanza(TipoPunto a, TipoPunto b) {
  float d;

  d = sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));

  return d;
}


/* main */

int main() {
  TipoPunto a, b;

                /* assegna valori al primo punto */
  a.x=0.324;
  a.y=1.34;

                /* assegna valori al secondo punto */
  b.x=0.0;
```

```
  b.y=0.0;

  printf("Distanza=%f\n", distanza(a, b));

  return 0;
}
```

Perimetro

Vediamo ora un altro esempio di uso della struttura punto. Supponiamo di avere un file di testo in cui sono memorizzate le coordinate di un insieme di punti, che possono essere al massimo cento. In particolare, su ogni linea ci sono due numeri reali, che rappresentano le coordinate di un punto. Questa sequenza di punti rappresenta una figura piana chiusa. Quello che vogliamo calcolare è il suo perimetro. Si noti che l'ultimo punto del file è da considerarsi unito al primo da un segmento.

Per prima cosa, analizziamo i dati del problema: abbiamo una sequenza di punti, per cui ci può sicuramente essere utile la definizione della struttura punto. Dal momento che il file contiene una intera sequenza di punti, possiamo leggere tutta la sequenza in memoria. Una sequenza di elementi si può rappresentare come un vettore. Dal momento che ogni elemento è un punto, usiamo un vettore di punti, ossia un vettore di strutture. La dichiarazione del vettore che usiamo per rappresentare l'insieme dei punti è:

```
  TipoPunto vett[100];
```

Nel dare questa definizione usiamo l'ipotesi che nel file ci siano al massimo cento punti. Se cosí non fosse, sarebbe stato necessario usare un array dinamico.

La prima cosa da fare è aprire il file, e da questo leggere i punti e memorizzarli nell'array di strutture. Questa prima parte non presenta difficoltà: si tratta di leggere un array da file, e quindi è analogo al programma di lettura di un array di interi da file. L'unica differenza è che il file contiene i dati di strutture invece che interi. Quindi, la lettura del singolo elemento non è più `res=fscanf(fd, "%d", &vett[n]);` (che legge un intero da file). Dal momento che la struttura è composta da due reali, possiamo usare, per leggere il singolo elemento del vettore:

```
    res=fscanf(fd, "%f %f", &vett[i].x, &vett[i].y);
```

Il resto del codice di lettura è uguale a quello del programma di lettura di array di interi da file. È possibile quindi usare un ciclo for per scandire gli elementi dell'array, oppure usare un ciclo while. Il valore di ritorno della funzione `fscanf` viene usato per capire quando il file è finito, ma può anche venire usato per capire se ci sono stati errori nella lettura (la cosa può avvenire per esempio se il file non è composto da coppie di reali, ma contiene per esempio dei caratteri che non si possono interpretare come numeri).

La parte più interessante del programma è il calcolo del perimetro. Dal momento che dobbiamo calcolare, per ogni coppia di punti consecutivi, la loro distanza, usiamo la funzione distanza definita nel programma precedente punti.c.

L'algoritmo che usiamo è il seguente: per ogni punto del vettore, calcoliamo la distanza fra questo punto e il successivo. Tutte queste distanze vengono sommate via via a una variabile che usiamo per memorizzare la somma parziale.

Ci sono ovviamente due cose da tenere in consderazione: la prima è che l'ultimo punto non ha un punto successivo nell'array. In altre parole, se `n` è la dimensione dell'array, allora non possiamo calcolare la distanza fra l'ultimo punto `vett[n-1]` e il successivo `vett[n]`, semplicemente perchè quest'ultimo non è definito (non è stato letto da file).

Usiamo quindi il seguente procedimento: per tutti i punti tranne l'ultimo, sommiamo la distanza fra il punto e il successivo. Alla fine del ciclo di somma, calcoliamo la distanza fra l'ultimo punto e il primo, e la aggiungiamo alla somma. Questo perchè abbiamo detto che la figura memorizzata su file va considerata chiusa, ossia l'ultimo punto è collegato al primo.

Il risultato di questo ragionamento è che il ciclo sull'array va fatto non su tutti gli indici che contengono un elemento significativo (che sono quelli da `0` a `n-1`) ma su tutti tranne l'ultimo elemento significativo, ossia si deve fare un ciclo in cui l'indice va da `0` a `n-2` (penultimo elemento significativo). Ad ogni passo si somma la distanza fra un punto e il successivo.

Il programma completo perimetro.c è qui sotto.

```
/*
  Definizione del tipo punto, lettura di
una sequenza di punti da file e memorizzazione
in un array, calcolo del perimetro della
figura chiusa definita dalla sequenza di punti.
*/

#include<stdlib.h>
#include<stdio.h>
#include<math.h>


/* definizione di struttura e tipo */

struct Punto {
  float x;
  float y;
};

typedef struct Punto TipoPunto;


/* funzione distanza */

float distanza(TipoPunto a, TipoPunto b) {
  float d;

  d = sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));

  return d;
}


/* main */

int main() {
  FILE *fd;
  int res;

  TipoPunto vett[100];
  int n;
  int i;
```

```
  float p;


                        /* apre il file */
  fd=fopen("punti.txt", "r");
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


                        /* lettura dei punti */
  for(i=0; i<=99; i++) {
    res=fscanf(fd, "%f %f", &vett[i].x, &vett[i].y);
    if( res!=2 )
      break;
  }
  n=i;


                        /* calcolo del perimentro */
  p=0;
  for(i=0; i<=n-1-1; i++)
    p+=distanza(vett[i], vett[i+1]);

  p+=distanza(vett[n-1], vett[0]);


                        /* stampa il perimetro */
  printf("Perimetro=%f\n", p);

  return 0;
}
```

Punti allineati

Un altro esercizio che può essere interessante risolvere è quello di scrivere una funzione che dice se tre punti su un piano cartesiano sono allineati.

Punto mediano

Il punto mediano fra due altri punti è quello che si trova esattamente in mezzo. Il calcolo delle coordinate del punto mediano si ottiene semplicemente facendo il calcolo della media fra le coordinate dei due punti.

La cosa interessante del programma risolutivo mediano.c è il fatto che una funzione può ritornare come risultato una variabile che è definita come una struttura. Questo si ottiene semplicemente usando il tipo struttura come tipo di ritorno della funzione, ossia si dichiara la funzione come: `TipoPunto mediano(TipoPunto a, TipoPunto b) {...`

```
/*
  Trova il punto mediano fra altri
due punti. La funzione che effettua
questo calcolo deve ritornare una
struttura punto.
*/
```

```
#include<stdlib.h>
#include<math.h>


/* definizione di struttura e tipo */

struct Punto {
  float x;
  float y;
};

typedef struct Punto TipoPunto;


/* punto mediano */

TipoPunto mediano(TipoPunto a, TipoPunto b) {
  TipoPunto m;

  m.x=(a.x+b.x)/2;
  m.y=(a.y+b.y)/2;

  return m;
}


/* main */

int main() {
  TipoPunto a, b, c;

                /* assegna valori al primo punto */
  a.x=0.324;
  a.y=1.34;

                /* assegna valori al secondo punto */
  b.x=0.0;
  b.y=0.0;


                /* trova il punto mediano */
  c=mediano(a, b);


                /* stampa le sue coordinate */
  printf("Punto mediano: (%f,%f)\n", c.x, c.y);

  return 0;
}
```

Strutture composte di strutture

Strutture composte di strutture

Supponiamo di voler rappresentare in memoria delle figure geometriche più complesse del punto, per esempio dei segmenti. Un segmento lo possiamo considerare composto dai suoi due punti estremi. Potremmo quindi usare due variabili separate, ognuna della quali rappresenta un punto. Come si è detto, ogni volta che due o più variabili servono per rappresentare lo stesso oggetto, bisogna prendere in considerazione l’opportunità di definire una struttura in modo tale da poter usare un singola

variabile composta di tipo struttura per rappresentare il singolo oggetto.

Nel nostro caso, un segmento è composto da due punti. Potremmo per esempio definire la struttura come composta da quattro valori reali. D'altra parte, il segmento è intuitivamente fatto di due punti, per cui il modo più naturale di definire la struttura è quelle di dire che è composta da due punti:

```
struct Segmento {
  TipoPunto primo;
  TipoPunto secondo;
};

typedef struct Segmento TipoSegmento;
```

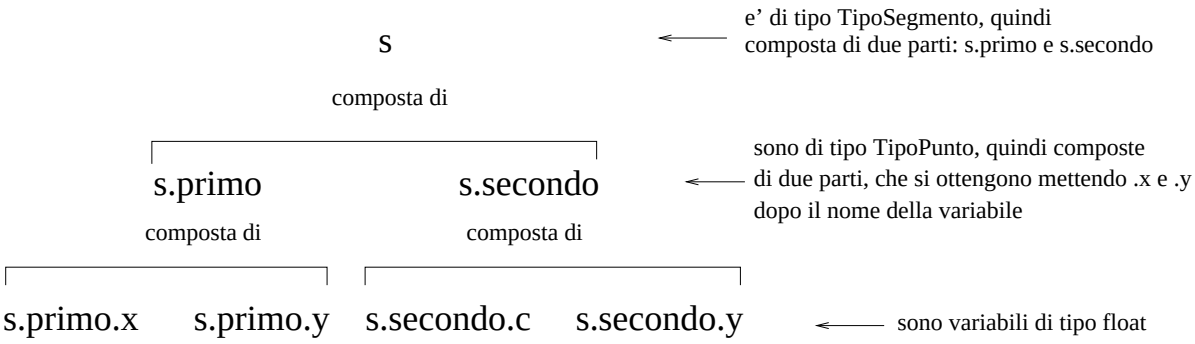
Il punto importante di questo esempio è che un tipo definito usando `struct` oppure `typedef` è un tipo che ha esattamente tutte le caratteristiche di un tipo predefinito. Dal momento che possiamo usare un tipo predefinito per indicare il tipo di un campo di una struttura, ne segue che possiamo usare allo stesso scopo anche un tipo che abbiamo definito noi. Nel nostro caso, la definizione:

```
struct Abcd {
  int primo;
  int secondo;
};
```

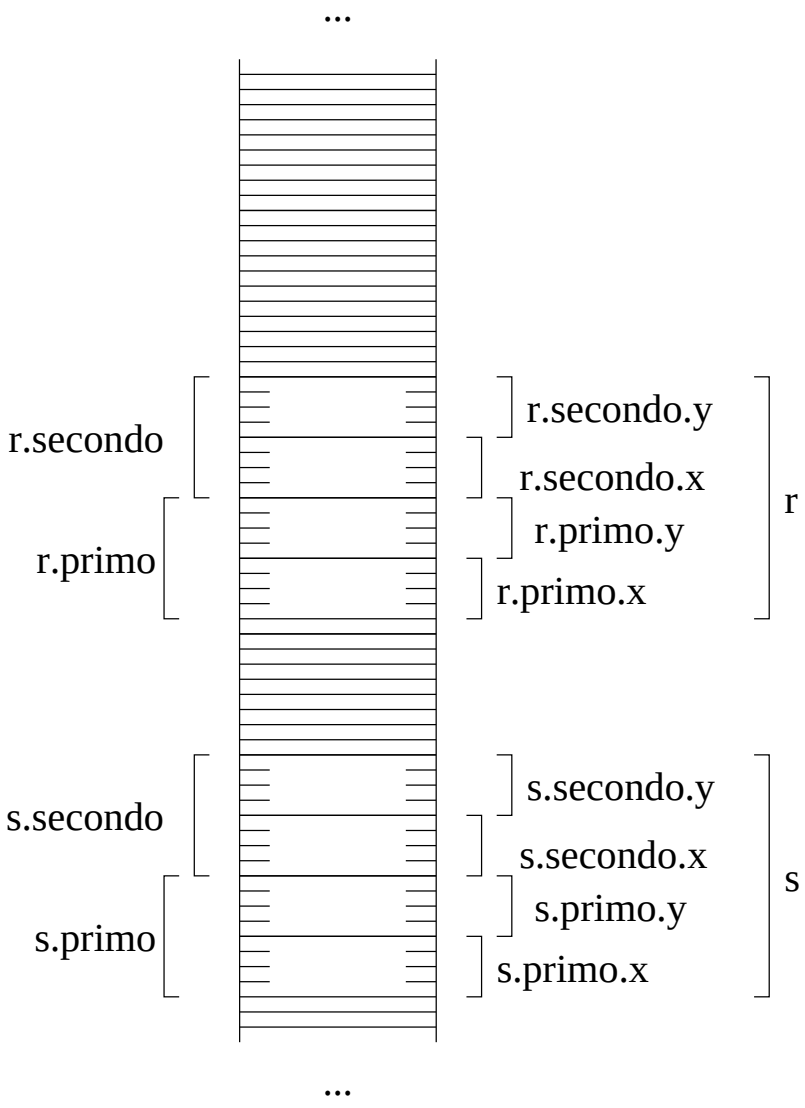
è una definizione valida. Dal momento che `TipoPunto` è un tipo definito, possiamo usarlo anche al posto di `int` nella definizione di struttura di sopra. In questo modo, abbiamo capito che la definizione della struttura segmento è una definizione valida. Questo discorso serve solo a far capire che definire una struttura in cui uno o più campi sono di tipo struttura, è perfettamente valido.

Questa nuova struttura definita con campi struttura è ancora una struttura come tutte le altre, per cui si può definire un tipo a partire da esso, definire delle variabili di questo tipo, passare la struttura a una funzione, ecc.

Il programma che segue definisce il tipo segmento, definisce una funzione che decide se due segmenti si intersecano, dichiara due variabili di questo tipo, assegna ad esse dei valori, e poi chiama la procedura per vedere se i due segmenti si intersecano. L'unico punto da capire è: in che modo si accede alle coordinate dei punti estremi del segmento? Se `s` è una variabile di tipo segmento, allora è composta da due sottovariabili `s.primo` e `s.secondo`, che sono entrambe variabili di tipo punto. Dal momento che `a.primo` è di tipo punto, è composta da due campi, a cui si può accedere mettendo `.x` e `.y` dopo il nome della variabile. Quindi, le coordinate del primo punto `s.primo` sono `s.primo.x` e `s.primo.y`. La figura seguente visualizza questo concetto.



La seguente figura fornisce una rappresentazione grafica della memoria occupata da due variabili di tipo segmento `s` e `r`.



Il programma completo segmenti.c è riportato qui sotto.

```
/*
  Definizione del tipo punto,
funzione che calcola la distanza.
*/
```

```
#include<stdlib.h>
#include<math.h>


/* definizione della struttura e tipo per il punto */

struct Punto {
  float x;
  float y;
};

typedef struct Punto TipoPunto;


/* definizione di struttura e tipo per il segmento */

struct Segmento {
  TipoPunto primo;
  TipoPunto secondo;
};

typedef struct Segmento TipoSegmento;


/* distanza fra due punti */

float distanza(TipoPunto a, TipoPunto b) {
  float d;

  d = sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));

  return d;
}


/* intersezione fra le due rette di due segmenti */

TipoPunto intersezione(TipoSegmento m, TipoSegmento n) {
  TipoPunto r;
  float a1, b1, c1;
  float a2, b2, c2;

  return r;
}


/* verifica se due segmenti si intersecano */

int intersecano(TipoSegmento m, TipoSegmento n) {
  TipoPunto i;

  i=intersezione(m, n);

  if( i.x<=m.primo.x && i.x>=m.secondo.x &&
      i.y<=m.primo.y && i.y>=m.secondo.y )
    return 1;
  else
    return 0;
}
```

```
/* main */

int main() {
  TipoSegmento s, r;

                /* assegna valori al primo segmento */
  s.primo.x=12.1;
  s.primo.y=1.2;
  s.secondo.x=-12;
  s.secondo.y=-1.2;

                /* assegna valori al secondo segmento */
  r.primo.x=-1.2;
  r.primo.y=0.0;
  r.secondo.x=.9;
  r.secondo.y=-.5;

  return 0;
}
```

Nomi di file

Nomi di file

Struttura che rappresenta dei nomi di file, insieme ad alcune loro caratteristiche (per esempio, la dimensione).