Puntatori

Prof. Orazio Mirabella

Definizione di Puntatore

Ogni variabile è caratterizzata da 4 proprietà:

- *Esempio:* **int a = 5;**

indirizzo	memoria
A00E	…
A010	**5**
A012	…
A014	…

Proprietà della variabile a:

➔ nome: a

➔ tipo: int

➔ valore: 5

➔ indirizzo: A010

Finora abbiamo usato solo le prime tre proprietà

- **&a -> operatore indirizzo** "&" applicato alla variabile a contiene il valore 0xA010 (ovvero, 61456 in decimale).

- Un indirizzo può essere assegnato solo a una speciale categoria di variabili dette *puntatori.*

Variabile puntatore

- Gli indirizzi si utilizzano nelle variabili di tipo puntatore, dette anche **puntatori**.

*Sintassi: tipo_base *var_punt*;*

- *Esempio:* **int *pi;**
- Proprietà della variabile pi:

➔ nome: pi

➔ tipo: **puntatore ad intero** (ovvero, indirizzo di un intero)

➔ valore: inizialmente casuale

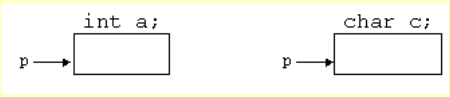
➔ indirizzo: fissato una volta per tutte

- in sostanza *var_punt* è creata per poter mantenere l'indirizzo di variabili di tipo *tipo_base*

Variabile puntatore

- Il tipo puntatore è un classico esempio di tipo derivato; infatti, non ha senso parlare di tipo puntatore in generale, ma occorre sempre specificare a quale tipo esso punta.

```
int a;
char c;
int *pi;
char *pc;
pi = &a;
pc = &c;
```



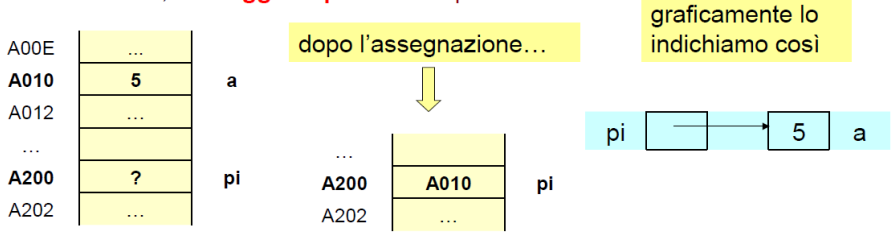
- L'indirizzo di una variabile puntatore viene fissato al momento della sua definizione, ma non il suo valore.
- Le variabili pi e pc sono inizializzate rispettivamente con l'indirizzo di a e di c.

Variabile puntatore

- Una variabile puntatore può essere inizializzata usando l'operatore di indirizzo.

Esempio: `pi = &a;`

- il valore di pi viene posto pari all'indirizzo di a
- ovvero, pi **punta** ad a
- ovvero, a è l'**oggetto puntato** da pi



Variabile puntatore

- L'operatore *, detto *operatore di indirezione*, si applica a una variabile di tipo puntatore e restituisce il contenuto dell'oggetto puntato
- Sia:

  a = 5;

  c = 'x';
- in memoria abbiamo la situazione illustrata di seguito.

- Le istruzioni :

  printf("a = %d c = %c", a, c);

  printf("a = %d c = %c", *pa, *pc);

hanno esattamente lo stesso effetto, quello di visualizzare:

a = 5 c = x

Variabile puntatore

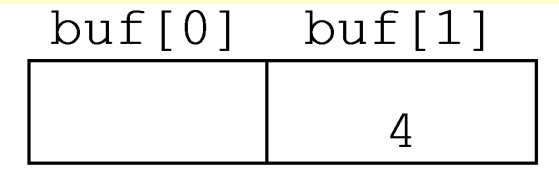
- Esempio:

```
char c1, c2;
char *pc;
...
c1 = 'a';
c2 = 'b';
printf(" c1 = %c, c2 = %c \n", c1, c2);
pc = &c1; /* pc contiene l'indirizzo di c1 */
c2 = *pc; /* c2 contiene il carattere 'a' */
printf(" c1 = %c, c2 = %c \n", c1, c2);
```

- Dopo l'assegnazione `pc=&c1`; i nomi `c1` e *pc sono perfettamente equivalenti (*alias*), e si può accedere allo stesso oggetto creato con la definizione char c1 sia con il nome c sia con l'espressione *pc
- L'effetto ottenuto con l'assegnazione c2=*pc si sarebbe ottenuto, equivalentemente, con l'assegnazione c2 = c1;

Variabile puntatore

- int buf[2];
- int *p;
- ...
- p = &buf[1];
- *p = 4;
- Con il puntatore a intero p e l'operatore * si è modificato il contenuto della locazione di memoria buf[1]



Array e Puntatori

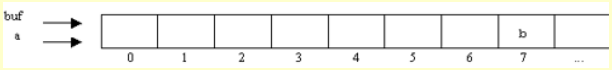
- Gli array e i puntatori in C sono strettamente correlati. Il nome di un array può essere usato come un puntatore al suo primo elemento

```
char buf[100];
char *s;
s = &buf[0];
s = buf;
```

- le due assegnazioni s=&buf[0] e s=buf sono perfettamente equivalenti. In C il nome di un array, è una costante il cui valore è l'indirizzo del primo elemento dell'array.

```
char buf[100];
char *s;
...
s = buf;
buf[7] = 'a';
printf("buf[7] = %c\n", buf[7]);
*(s + 7)= 'b';
printf("buf[7] = %c\n", buf[7]);
```

s e buf sono due sinonimi, con la differenza che s è una variabile puntatore a carattere, mentre buf è una costante.



Array e Puntatori

```
char buf[2];
for (i = 0; i < 2; i++)
buf[i] = 'K';
```

per l'inizializzazione degli elementi di un array col puntatore:

```
char *s;
char buf[2];
s = buf;
for (i = 0; i < 2; i++)
*s++ = 'K';
```

L'istruzione *s++='K' opera nel modo seguente:

· copia K nella locazione di memoria puntata da s

· poi incrementa s di un elemento.

Aritmetica dei puntatori

- Un puntatore contiene un indirizzo e le operazioni che possono essere compiute su un puntatore sono perciò quelle che hanno senso per un indirizzo:
  - ➔l'incremento, per andare da un indirizzo più basso a uno più alto
  - ➔il decremento, per andare da un indirizzo più alto a uno più basso

  Operatori: + ++ - --

  Cosa significa incrementare/decrementare un puntatore?

  Es. (ipotizziamo che pc valga 10)

```
int a[10];
char b[10];
int *pi;
char *pc;
pi = a;
pc = b;
pi = pi + 3;/*sposta in avanti di 3 posizioni int*/
pc = pc + 3;/*sposta in avanti di 3 posizioni char*/
```

Aritmetica dei puntatori

- in generale, dato un operatore aritmetico applicato a un puntatore p che punta a un elemento di un array, p+1 significa "prossimo elemento del vettore" mentre p-1 significa "elemento precedente".
- La sottrazione tra puntatori è definita solamente quando entrambi i puntatori puntano a elementi dello stesso array
- La sottrazione di un puntatore da un altro produce un numero intero corrispondente al numero di posizioni tra i due elementi dell'array.
- sommando o sottraendo un intero da un puntatore si ottenga ancora un puntatore.

Aritmetica dei puntatori

```
int v1[10];
int v2[10];
int i;
int *p;

i = &v1[5] - &v1[3]; /* 1 ESEMPIO */
printf("%d\n", i); /* i vale 2 */

i = &v1[5] - &v2[3]; /* 2 ESEMPIO */
printf("%d\n", i); /* il risultato è indefinito */

/* 3 ESEMPIO */
p = v2 - 2; /* dove va a puntare p ? */
```

Passaggio di parametri per indirizzo

- All'apparenza le funzioni sembrano essere limitate dal meccanismo del passaggio parametri per valore.
- Il programmatore C risolve questa apparente pecca passando per valore l'indirizzo della variabile – array o altro.
- Passare un indirizzo a una funzione significa renderle nota la locazione dell'oggetto corrispondente all'indirizzo.
- Questo meccanismo è noto con il nome di *passaggio di parametri per indirizzo*.

```
#include <stdio.h>
void scambia(int, int);
main()
{
int x, y;
x = 8;
y = 16;
printf("Prima dello scambio\n");
printf("x = %d, y = %d\n", x, y);
scambia(x, y);
printf("Dopo lo scambio\n");
printf("x = %d, y = %d\n", x, y);
}
/* Versione KO di scambia */
void scambia(int a, int b)
{
int temp;
     temp = a;
     a = b;
     b = temp;
}
```

N.b.:la chiamata scambia(x, y); non ha effetto sulle variabili intere x e y. Infatti i valori di x e y sono copiati nei parametri formali a e b e, quindi, sono stati scambiati i valori dei parametri formali, non i valori originali di x e y.

```
#include <stdio.h>
void scambia(int *, int *);
main()
{
int x, y;
x = 8;
y = 16;
printf("Prima dello scambio\n");
printf("x = %d, y = %d\n", x, y);
scambia(&x, &y);
printf("Dopo lo scambio\n");
printf("x = %d, y = %d\n", x, y);
}
/* Versione OK di scambia */
void scambia(int *a, int *b)
{
int temp;
     temp = *a;
     *a = *b;
     *b = temp;
}
```

```
#include <stdio.h>
void scambia(int, int);
main()
{
int x, y;
x = 8;
y = 16;
printf("Prima dello scambio\n");
printf("x = %d, y = %d\n", x, y);
scambia(x, y);
printf("Dopo lo scambio\n");
printf("x = %d, y = %d\n", x, y);
}
/* Versione KO di scambia */
void scambia(int a, int b)
{
int temp;
     temp = a;
     a = b;
     b = temp;
}
```

Passaggio di un array a una funzione

```
#include <stdio.h>
char str[] = "BATUFFO";
int strlen(char *);
main()
{
printf("la stringa %s è lunga %d\n", str, strlen( str ));
}
int strlen( char *p)
{
int i = 0;
while (*p++) i++;
return i;
}
```

Funzioni della libreria <string.h>

char *strcat(char **string1*, const char **string2*);

Concatena le stringhe *string1* e *string2* attaccando *string2* in coda a *string1*.

char *strncat(char **string1*, const char **string2*, int n);

Concatena le stringhe *string1* e *string2* attaccando *n* caratteri della stringa *string2* in coda a *string1*.

int strcmp(const char **string1*, const char **string2*);

Confronta *string1* con *string2*. Ritorna 0 se le stringhe sono identiche, un numero minore di zero se *string1* è minore di *string2*, e un numero maggiore di zero se *string1* è maggiore di *string2*.

char *strcpy(char **string1*, const char **string2*);

Copia *string2* su *string1*.

char *strncpy(char **string1*, const char **string2*, int n);

Copia i primi *n* caratteri di *string2* su *string1*.

int strlen(const char **string*);

Conta il numero di caratteri di *string*, escluso il carattere nullo.

char *strchr(const char **string*, int c);

Ritorna il puntatore alla prima occorrenza in *string* del carattere c.

char *strrchr(const char **string*, int c);

Ritorna il puntatore all'ultima occorrenza del carattere c nella stringa *string*.

Oggetti dinamici

- Mentre gli oggetti statici vengono creati specificandoli in una definizione, gli oggetti dinamici sono creati/distrutti durante l'esecuzione del programma, non durante la compilazione.
- Il valore NULL, che può essere assegnato a qualsiasi tipo di puntatore, indica che nessun oggetto è puntato da quel puntatore.
- in C la memoria è allocata dinamicamente per mezzo delle funzioni di allocazione malloc e calloc che hanno le seguenti specifiche:
- void *malloc(int num); /* num: quantità di memoria da allocare */
- void *calloc(int numele, int eledim);
- /* numele: numero di elementi;
- eledim: quantità di memoria per ogni elemento */
- Sia malloc sia calloc ritornano un puntatore a carattere che punta alla memoria allocata.

Memoria richiesta

- Per stabilire la quantità di memoria da allocare è molto spesso utile usare l'operatore sizeof, che si applica nel modo seguente:
- sizeof(*espressione*) :restituisce la quantità di memoria richiesta per memorizzare *espressione*, oppure
- sizeof(*T*) : restituisce la quantità di memoria richiesta per valori di tipo *T*.
- Es:

```
main()
{
char a[10];
int i;
i = sizeof(a[10]);
printf("L'array %s ha dimensione = %d\n", a, i);
i = sizeof(int);
printf("Gli interi hanno dimensione %d", i);
}
```

- L'operatore sizeof ritorna un intero maggiore di zero corrispondente al numero di char che formano l'espressione o il tipo.