Struttura di un programma e classi di memoria

Struttura del codice (1)

- Un programma è formato da una o più unità di compilazione separate (file).
  - Le **definizioni** di funzioni e variabili globali devono essere uniche nell'intero programma.
    - Deve esistere una funzione `main`.
- Le varie unità di compilazione possono essere compilate separatamente.
  - I file oggetto generati vengono poi aggregati dal linker a formare un unico eseguibile.

Struttura del codice (2)

- Il programmatore è responsabile di garantire la coerenza tra le dichiarazioni la definizione della stessa entità, se si trovano in unità di compilazione separate.
  - Il compilatore non effettua controlli.

La funzione main (1)

- E’ la funzione dalla quale inizia l’esecuzione del programma.
- Ha prototipo:

```
int main(void);
```

  oppure

```
int main(int argc, char* [] argv);
```

- Il primo prototipo vale in ambienti freestanding, il secondo in ambienti hosted.

La funzione main

- Il valore prodotto (di tipo `int`) viene passato al sistema operativo, come valore di terminazione del programma.
- Non deve essere dichiarata, solo definita.
- Non può essere definita `inline`.

Uscita dal programma

- Da un programma C si esce in due modi:
  - con un `return` dal `main`;
  - eseguendo la funzione `exit`, con valore di terminazione del programma come argomento.
- Il valore di terminazione del programma deve essere:
  - `EXIT_SUCCESS`, per indicare la terminazione normale;
  - `EXIT_FAILURE`, per indicare la terminazione in errore.
  - Le due costanti sono definite in stdlib.h.

Esempio di programma (1)

- Legge un file in input e scrive tutte e sole le righe contenenti un certo pattern.
  - Si ipotizza una lunghezza massima di 1000 caratteri per riga.

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

  // Lunghezza massima della linea
# define  MAXLINE   1000
```

Esempio di programma (2)

```
bool getline(char line [],
             unsigned int max);
bool contains(const char source [],
              const char searchfor []);

  // pattern da cercare
const char pattern [] = "test";
```

Esempio di programma (3)

```
int main(void)
  {
  char            line [MAXLINE];
  unsigned long   found = 0;

  while (getline(line, MAXLINE))
        if (contains(line, pattern))
              {
              printf("%s", line);
              ++found;
              }
  exit(found > 0? EXIT_SUCCESS: EXIT_FAILURE);
  }
```

Esempio di programma (4)

```
bool getline(char line [], unsigned int max)
  {
  int             c;
  unsigned int    i;
  i = 0;

  while (--max > 0 && (c = getchar()) != EOF)
       if ((line [i++] = c) == '\n')
            {
            line [i] = '\0';
            return true;
            }
  return false;
  }
```

Esempio di programma (5)

```
bool contains(const char source [],
  const char searchfor []);
   {
   unsigned int   i, j, k;

   for (i = 0; source [i] != '\0'; ++i)
  {
  for (j = i, k = 0; source [j] == searchfor [k] &&
        source [j] != '\0' && searchfor [k] != '\0';
        ++j, ++k);
        if (searchfor [k] == '\0')
              return true;
        }
   return false;
   }
```

Classi di memoria di una variabile esterna

- Per una variabile dichiarata **fuori** dalle funzioni:
  - default: definita e accessibile da altre unità di compilazione;
  - `extern`: dichiarata, non definita (definita altrove), resa accessibile dalla dichiarazione;
  - `static`: definita e non accessibile da altre unità di compilazione.

Classi di memoria di una variabile interna

- Per una variabile dichiarata **dentro** una funzione quindi comunque **inaccessibile da altre funzioni**:
  - `auto` (default): variabile automatica;
  - `register`: variabile automatica, da allocare in un registro, se possibile;
  - `extern`: dichiarata, non definita (definita altrove), resa accessibile dalla dichiarazione;
  - `static`: variabile statica.

Classi di memoria di una funzione

- In una dichiarazione possono essere:
  - `extern` (default): definita altrove, resa accessibile dalla dichiarazione;
  - `static`: non accessibile da altre unità di compilazione.
- In una definizione possono essere:
  - (default): accessibile da tutte le unità di compilazione;
  - `static`: non accessibile da altre unità di compilazione.

Esempio di extern

- File_1.c:

```
extern unsigned int counter;
extern void f(void);
…
```

- File_2.c:

```
unsigned int counter;
void f(void)
  {
  …
  }
```

Esempio di static

```
int             v1;
static int      v2;

void f1(void)
  {
  …
  }
static void f2(void)
  {
  …
  }
```

Accessibile da altri file

Non accessibile da altri file

Accessibile da altri file

Non accessibile da altri file

Esempio di variabili locali

```
void f(void)
  {
  int              v1;
  auto int         v2;
  register int     v3;

  static int       v4;
  …
  }
```

Variabile automatica

Variabile automatica

Variabile automatica, allocata in un registro, se possibile

Variabile statica

Funzioni e variabili esterne

- Se dichiarate `static`, sono visibili
  - Nel loro scope (dalla dichiarazione alla fine dell'unità di compilazione).
- Se non dichiarate `static`, sono visibili
  - Nel loro scope (dalla dichiarazione alla fine dell'unità di compilazione).
  - In ogni altra unità di compilazione, se ivi dichiarate `extern`.

Esempio di variabile locale statica

```
void f(void)
  {
  static unsigned long counter = 0;

  ++counter;
  …
  }
```

La variabile `counter` conserva il valore attraverso differenti chiamate, quindi può essere usata per contare le chiamate alla funzione.

Significati di static

- Per funzioni e variabili esterne alle funzioni, limita la visibilità all'unità di compilazione in cui si trova.
  - In altre unità di compilazione potranno trovarsi omonimi senza conflitto.
- Per variabili interne alle funzioni, le rende statiche, cioè allocate permanentemente e non deallocate all'uscita del blocco.
  - In altre unità di compilazione o in altre funzioni potranno trovarsi variabile omonime senza conflitto.
  - Come tutte le variabili statiche, possono essere inizializzate solo con costanti.

Significato di register

- “Consiglia” al compilatore di allocare la variabile in un registro, perché usata molte volte.
  - applicabile solo a variabili interne automatiche e argomenti di una funzione.
- Il compilatore accetta o meno il consiglio, in funzione del tipo e dei registri disponibili.
  - Decide caso per caso.
  - Il consiglio è solitamente accettato solo per i tipi predefiniti.
  - Dichiarazioni `register` in eccesso sono ignorate, e sono innocue.

Vincoli sulle variabili register

- Non si può avere un puntatore a una variabile `register`, anche se il consiglio è stato ignorato.
  - Il compilatore non rende nota **in nessun modo** la scelta fatta.

Sintesi delle classi di memoria

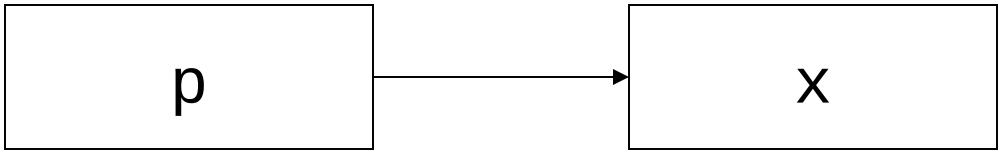
Classe di memoria	Variabile interna	Variabile esterna	Funzione
`extern`	Definita altrove	Definita altrove	Definita altrove
`static`	Allocata in permanenza	Visibile solo nell'unità di compilazione	Visibile solo nell'unità di compilazione
`auto`	Automatica		
`register`	Allocata in un registro		
Nessuna	Come `auto`	Visibile nelle altre unità di compilazione	Visibile nelle altre unità di compilazione

Puntatori

Puntatori

- Un puntatore è un "riferimento" a un altro oggetto (variabile o funzione).
  - Generalmente implementato tramite l'indirizzo dell'oggetto puntato.
  - Esempio:

```
int     x;
int*    p = &x;
```



Operatori per i puntatori

- Con i puntatori si utilizzano due operatori, unari prefissi:

  `&`, che produce come valore il puntatore all'operando, che deve essere un **lvalue** non dichiarato `register`.

  `*`, che accede all'oggetto puntato dall'operando, che deve essere un puntatore.

  - Utilizzato anche come costruttore di tipo, per dichiarare puntatori.

Esempi di puntatori

```
int      i, j;
int*     p;

p = &i;  // p punta a i
*p = 5;  // equivale a i = 5;
j = *p   // equivale a j = i;
```

I puntatori

- **Non sono indirizzi macchina**!
- Sono un modo di accedere a una variabile.
  - L'indirizzo macchina è l'implementazione **più frequente**, ma **non l'unica**.
- In generale puntatori a tipi diversi non sono compatibili, né convertibili.
  - Possono persino avere dimensioni diverse!

Puntatori e tipi (1)

- In C i puntatori hanno un tipo, che viene verificato in compilazione:
  - un puntatore può puntare solo a oggetti del tipo dichiarato;
    - al massimo meno qualificati o con diverso modificatore (`signed` o `unsigned`);
  - un puntatore è compatibile solo con puntatori dello stesso tipo.
- Un puntatore può essere convertito a un puntatore di tipo diverso, mediante l'operatore cast, ma il risultato in generale dipende dall'implementazione.

Puntatori e tipi (2)

```
int*    pi;
int*    qi;
float*  pf;

pi = qi;          // OK
pf = pi;          // Errore
pf = (float*)pi; // ??
```

L'ultimo assegnamento è legale, ma gli effetti dell'uso di `pf` dipendono dall'implementazione.

Esempio di puntatori

- Simulazione del passaggio di argomenti per riferimento.

```
void    scambia(int* x, int* y)
   {
   int  temp;

   temp = *x;
   *x = *y;
   *y = temp;
   }

scambia(&a, &b);
```

Assegnamento tra puntatori

- E’ sempre possibile assegnare a un puntatore un valore **dello stesso tipo**.
  - Il valore può essere `NULL`.
- Le stesse regole valgono per il passaggio di argomenti e la produzione di valori da parte di funzioni.

Funzioni e puntatori

- Una funzione può produrre un puntatore.
  - Esempio di dichiarazione:

```
int* f(void);
```

Attenzione! Una funzione del genere non deve **mai** produrre come valore il puntatore a una sua **variabile locale**, perché dopo l'esecuzione della funzione punterebbe a un oggetto non più esistente.

Esempio di funzione che produce un puntatore (1)

- Funzione che produce il puntatore al massimo tra i primi n elementi di un vettore.

```
int v [10];
int* pmax(register unsigned int n)
   {
   register unsigned int max_index = 0;

   for (; n > 0; --n)
        if (v [n] > v [max_index])
             max_index = n;
   return &v [max_index];
   }
```

Esempio di funzione che produce un puntatore (2)

Implementazione alternativa.

```
int* pmax(register unsigned int n)
   {
   register unsigned int* pm = &v [0];

   for (; n > 0; --n)
        if (v [n] > *pm)
             pm = &v [n];
   return pm;
   }

// azzera il massimo tra i primi 4
*pmax(4) = 0;
```

Puntatori generici

- Puntano a oggetti di qualsiasi tipo.
  - In pratica sfuggono ai normali controlli di tipo.
    - Questo non autorizza però a effettuare operazioni arbitrarie su di essi.
- Si indicano col tipo `void *`.
- Il puntatore va convertito al tipo corretto, tramite cast, prima di essere utilizzato.
- Qualsiasi puntatore può essere convertito a `void *` e indietro **al tipo originale**.

Esempi di puntatori generici

```
int      i;
int*     pi;
void*    pv;
```

`pi = &i;`	Normale e corretto
`pv = pi;`	OK, conversione automatica
`pv = (void *)pi;`	Meglio!
`*(int *)pv = 0;`	Corretto! azzera `i`
`*(char *)pv = 0;`	Comportamento indefinito!

Puntatore nullo

- Indicato con `NULL`, di tipo `void *`.
  - Definito come `(void *)0`.
  - Non punta a nessun oggetto.
  - E’ diverso da qualsiasi puntatore a variabile o funzione.

Esempi di puntatore nullo

```
char*     p;

p = NULL;                   Conversione implicita
p = (void*)0;               Conversione implicita
p = (char*)(void*)0;        Corretto, ma barocco
p = (char*)NULL;            Il più chiaro
p = 0;                      Illegale!
```

Conversioni tra puntatori (1)

- Automatiche da `void *` ad altro tipo puntatore e viceversa in:
  - assegnamenti;
  - confronti;
  - passaggio di argomenti;
  - `return`.
- E' comunque preferibile specificare le conversioni con un cast.

Conversioni tra puntatori (2)

- Le uniche conversioni che hanno un comportamento definito e portabile sono:
  - da `NULL` a qualsiasi tipo puntatore;
  - da un puntatore prodotto da funzioni per la gestione della memoria dinamica (`malloc`) a qualsiasi tipo puntatore, ma non puntatore a funzione;
  - da puntatore qualsiasi a `void *`;
  - da `void *` al **tipo originale**;
  - da puntatore qualsiasi a `char *`;
  - conversioni che **aggiungono** qualificatori.
- Ogni altra conversione tra puntatori **è un errore** e non è portabile.

Esempi di conversioni di puntatori

```
struct s { … } *p;
void*     v;

p = (struct s*)NULL;
p = (struct s*)malloc(sizeof(struct s));

v = (void*)p;
v = realloc(v, sizeof(struct s) * n);
p = (struct s*)v;
```

oppure:

```
p = (struct s*)realloc((void *)p,
  sizeof(struct s) * n);
```

Concisione e chiarezza

- `if (p)` è legale, perché un puntatore è “vero” se diverso da `NULL`, ma

  `if (p != NULL)` è decisamente preferibile.

  Meglio ancora `if (p != (`<*tipo*>` *)NULL)`
- Privilegiare sempre la chiarezza!

Una trappola lessicale

Attenzione!

```
y = x/*p    /* p punta al divisore */;
```

Equivale a:

```
y = x/*p    /* p punta al divisore */;
```

Commento!

- **Separare sempre** gli operatori binari con spazi!
- **Non mettere mai commenti entro un'istruzione**, ma solo alla fine.

Vettori e matrici

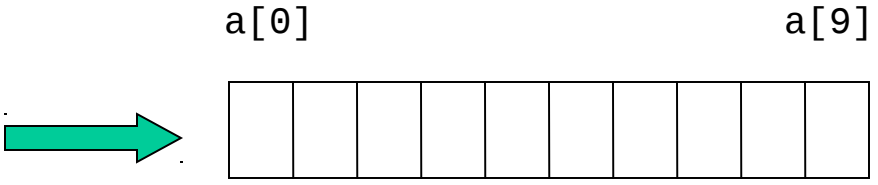
Vettori

- Aggregati **ordinati** di elementi **dello stesso tipo**.
  - Ogni vettore si comporta come se fosse allocato a indirizzi di memoria consecutivi.
- L’accesso avviene tramite indici numerici a partire da zero.

Dichiarazione di vettori

- Si dichiarano come le altre variabili.
- Al nome segue il numero di elementi, tra parentesi quadre.
  - P. es.:

```
int a [10];
```

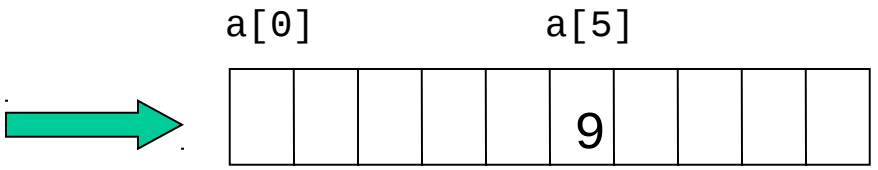


- Il numero di elementi deve essere **costante** per i vettori **statici**, può essere un’espressione per i vettori automatici.

Accesso ai vettori

- Al nome segue l'indice, tra parentesi quadre.
  - P. es.:

```
a [5] = 9;
```



- L'indice deve essere un'espressione intera, con valore tra zero e il numero di elementi meno uno.
  - Non vi sono controlli e in caso di errore il comportamento è indefinito.

Inizializzazione di vettori (1)

- Si inizializzano come le variabili, ma con una lista di valori separati da virgole e racchiusi tra parentesi graffe.
  - P. es.:

```
int a [5] = { 0, 1, 2, 3, 4 };
```

- Se i valori sono troppi, si ha un errore.
- Se i valori sono pochi, gli elementi mancanti sono inizializzati a zero.
- I valori tra graffe sono convertiti al tipo base del vettore con le regole degli assegnamenti.

Calcolo della dimensione dei vettori.

- Nella dichiarazione si può omettere il numero di elementi di un vettore, se lo stesso è inizializzato.
  - Il compilatore deduce il numero di elementi dalle inizializzazioni.

  Per esempio:

```
int a [] = { 0, 1, 2, 3, 4 };
```

  Equivale a:

```
int a [5] = { 0, 1, 2, 3, 4 };
```

Dimensione di vettori (1)

- La dimensione di un vettore è sempre **uguale** al prodotto tra la dimensione di un elemento e il numero di elementi.
- Pertanto

  `sizeof(`<*vettore*>`) / sizeof(`<*elemento*>`)`

  permette di calcolare il numero di elementi di un vettore.
  - E' indispensabile se la dimensione del vettore non è stata specificata.
  - **Non funziona** con i vettori passati come **argomenti**.

Dimensione di vettori (2)

- Per esempio:

```
int v [] = { 2, 3, 5, 7, 11 };
```

`sizeof(v) / sizeof(v [0])` dà `5`.

Dimensione di vettori (3)

- Un ciclo della forma

```
for (i = 0; i < sizeof(v) /
        sizeof(v [0]); ++i)
```

è preferibile a un ciclo come:

```
for (i = 0; i < MAX_ELEMENTS; ++i)
```

Perché più chiaro e indipendente dalla dichiarazione del vettore.

Naturalmente cicli come:

```
for (i = 0; i < 100; ++i)
```

Rappresentano una forma di programmazione infantile.

Numero di elementi dei vettori

- Può essere calcolato con l'operatore `sizeof` anche per vettori di dimensione variabile.
  - P. es. scrivendo:

```
int    f(size_t n)
  {
  int  b [n + 3];
```

  con `sizeof(b) / sizeof(b [0])` si calcola il numero di elementi del vettore `b`.

Matrici (1)

- Una matrice è un vettore a più dimensioni.
  - Nella dichiarazione e nell'utilizzo si usa una coppia di parentesi quadre per ogni dimensione.

  Per esempio:

```
int a [4] [5];
a [i] [j] = i * j;
```

- Il numero di massimo di dimensioni è almeno 7 e dipende dall'implementazione.
- Comunque sono accettate più dimensioni di quante servano in pratica.
  - Matrici con più di 3 dimensioni sono molto rare.

Matrici (2)

- Nella inizializzazione si deve usare un livello di graffe per ogni dimensione.

  Per esempio:

```
int a [2] [3] =
      {
          { 1, 2, 3 },
          { 4, 5, 6 }
      };
```

- Come al solito, gli elementi mancanti sono inizializzati a zero.

Matrici (3)

- Nella dichiarazione, in presenza di inizializzazione, si può omettere solo il **primo** indice.

  Per esempio:

```
int a [] [3] =
   {
       { 1, 2, 3 },
       { 4, 5, 6 }
   };
```

Matrici (4)

- Nell'inizializzazione è ammesso usare un solo livello di parentesi graffe (prassi **da evitare** assolutamente).

  Per esempio:

```
int a [2] [3] =
        { 1, 2, 3, 4, 5, 6 };
```

Allocazione di matrici

- L’allocazione è per righe: gli elementi si susseguono riga per riga.

  Per esempio, `a [2] [3]` è memorizzata nell’ordine:

  `a [0] [0]`, `a [0] [1]`,
  `a [0] [2]`, `a [1] [0]`,
  `a [1] [1]`, `a [1] [2]`.

Vettori e puntatori

Il nome del vettore

- A tutti gli effetti il nome di un vettore è una **costante** di tipo puntatore al tipo base del vettore. `a` è **sinonimo** di `&a [0]`.

  Quindi è illegale scrivere:

  ```
  int  a[10], b[10];
  a = b;
  ```

- In C, a differenza che in altri linguaggi (Pascal, Ada), non c'è modo di assegnare un vettore a un altro con un'unica operazione, tranne che mediante funzioni di libreria (`memcpy, memmove`).

Vettori e puntatori

- Puntatori e vettori sono intercambiabili nell'uso:
  - A tutti gli effetti, se `p` è un puntatore:

  `*p` equivale a `p [0]`;

  `*(p + n)` equivale a `p [n]`.
- Si usano comunemente puntatori per accedere agli elementi di un vettore.

Esempio di uso di vettori e puntatori

- Dichiarando:

```
int     a [10];
int*    p = a;
```

Le seguenti notazioni sono del tutto equivalenti:

Vettori e indici	vettori e *	puntatori e indici (offset)	puntatori e *
`&a [0]`	`a`	`p`	`p`
`&a [i]`	`a + i`	`&p [i]`	`p + i`
`a [0]`	`*a`	`p [0]`	`*p`
`a [i]`	`*(a + i)`	`p [i]`	`*(p + i)`

Notazione corretta

- Per chiarezza, si raccomanda l'uso dell'operatore * con i puntatori e delle parentesi quadre con i vettori.

Operazioni ammesse tra puntatori e interi

- Sommando o sottraendo un intero *n* a un puntatore lo si porta a puntare all'elemento *n*-esimo seguente o precedente.
- Il risultato è definito se e solo se punta ancora **nello stesso vettore** o all'elemento (inesistente) **immediatamente dopo** la fine del vettore.

Esempi di operazioni tra puntatori e interi

```
int      a [10];
int*     p = &a [3];
```


`p + 1` punta a `a [4]`


`p - 3` punta a `a [0]`


`p += 1` punta a `a [4]` e modifica `p`


`p -= 3` punta a `a [0]` e modifica `p`


`++p` modifica `p`, che punta a `a [4]`


`--p` modifica `p`, che punta a `a [2]`

Compatibilità tra puntatori

- Un puntatore è compatibile (nei confronti e nelle sottrazioni) solo con un puntatore **dello stesso tipo**, eventualmente con diverso **modificatore** (`signed` o `unsigned`) o **qualificatore** (`const` o `volatile`).
- Operazioni tra puntatori non compatibili danno risultati indefiniti.

Confronti tra puntatori (1)

- E’ ammesso il confronto per uguaglianza o diversità, tra puntatori **compatibili** o tra un puntatore e `NULL`.
  - Due puntatori sono uguali se puntano alla stessa variabile o sono entrambi `NULL`.
- Il risultato è definito se e solo se:
  - uno dei puntatori è `NULL`;
  - i due puntatori puntano a entità (variabili o funzioni) **dello stesso tipo**.
- Puntatori a **entità distinte** di tipo diverso **possono apparire uguali** (per esempio un puntatore a funzione e un puntatore a variabile).

Confronti tra puntatori (2)

- E’ ammesso il confronto in genere tra puntatori **compatibili**, che puntino **nello stesso vettore**.
  - E’ maggiore il puntatore che punta all’elemento con indice maggiore.
- Il risultato è definito se e solo se:
  - i due puntatori puntano **nello stesso vettore** o all'elemento **immediatamente seguente** al vettore;
  - i due puntatori puntano a campi di una **stessa struttura.**
- Il confronto tra puntatori a **entità distinte** dà valore indefinito.

Differenze tra puntatori (1)

- Si possono sottrarre puntatori **compatibili** che puntino **nello stesso vettore**.
  - Il risultato è il numero di elementi compresi tra i due puntatori, di tipo `ptrdiff_t`, definito nel file stddef.h come tipo intero `signed`; (`ptrdiff_t` è generalmente `int`).
- Il risultato è definito se e solo se i due puntatori puntano **nello stesso vettore** o all'elemento **immediatamente seguente** al vettore;
- La sottrazione tra puntatori a **entità distinte** dà valore indefinito.

Esempi di operazioni tra puntatori

```
int      a [10];
int*     p = &a [3];
int*     q = &a [5];
```

```
p == q   ⟹  falso
p >= q   ⟹  falso
p < q    ⟹  vero
p - q    ⟹  -2
q - p    ⟹  2
```

Esempio di uso di matrici (1)

```
# include <stdio.h>

int a [10] [5];

int main(void)
  {
  register  unsigned int   irig, icol;
  register  int            cont;

        // puntatore agli elementi
  register  int* p;
```

Esempio di uso di matrici (2)

```
   // inizializzazione tabella tramite
   // indici
for (irig = 0; irig < 10; ++irig)
   for (icol = 0; icol < 5; ++icol)
        a [irig] [icol] = cont++;

   // stampa della tabella tramite
   // puntatore a elemento
 for (p = &a [0][0]; p < &a [10][0]; ++p)
   printf("%d\n", *p);
 }
```

Esempio di uso di matrici (3)

```
  // Alternativa: stampa della
  // tabella tramite puntatore a riga

register int   (*prig) [5]
register int*  pcol;

for (prig = ((int [5])*)a [0];
     prig < ((int [5])*)a [5]; ++prig)
     for (pcol = &(*prig)[0];
          pcol < &(*prig)[5]; ++pcol)
          printf ("%d\n", *pcol);
```

Aritmetica dei puntatori (1)

- E’ coerente e indipendente dal tipo dell’oggetto puntato.
  - Per esempio, quale che sia il tipo:

<*tipo*> `a [10];`
<*tipo*> `* p = &a [3];`
<*tipo*> `* q = &a [5];`
`p + 2 == q` e `q – p = 2`

- Il compilatore tiene conto delle dimensioni degli oggetti; il programmatore non deve pensare in termini di indirizzi macchina, ma di **elementi del vettore**.

Aritmetica dei puntatori (2)

- L’integrazione tra vettori e puntatori e l’aritmetica degli indirizzi sono punti di forza del C, ma vanno sapute utilizzare e non sono prive di rischi.
- I puntatori possono essere lievemente più efficienti degli indici per accedere a un vettore.
  - Nei casi semplici il compilatore provvede alla conversione.

Attenzione a incrementi e decrementi

- Autoincremento e autodecremento modificano **prima** dell'uso se prefissi, **dopo** se postfissi.
- Pertanto, se `p` punta a `a [5]`:

  `*++p` incrementa `p` e accede a `a [6]`;

  `++*p` incrementa `a [5]` e vi accede;

  `*p++` accede a `a [5]` e poi incrementa `p`;

  `(*p)++` accede a `a [5]` e poi lo incrementa;

  `*--p` decrementa `p` e accede a `a [4]`;

  `--*p` decrementa `a [5]` e vi accede;

  `*p--` accede a `a [5]` e poi decrementa `p`;

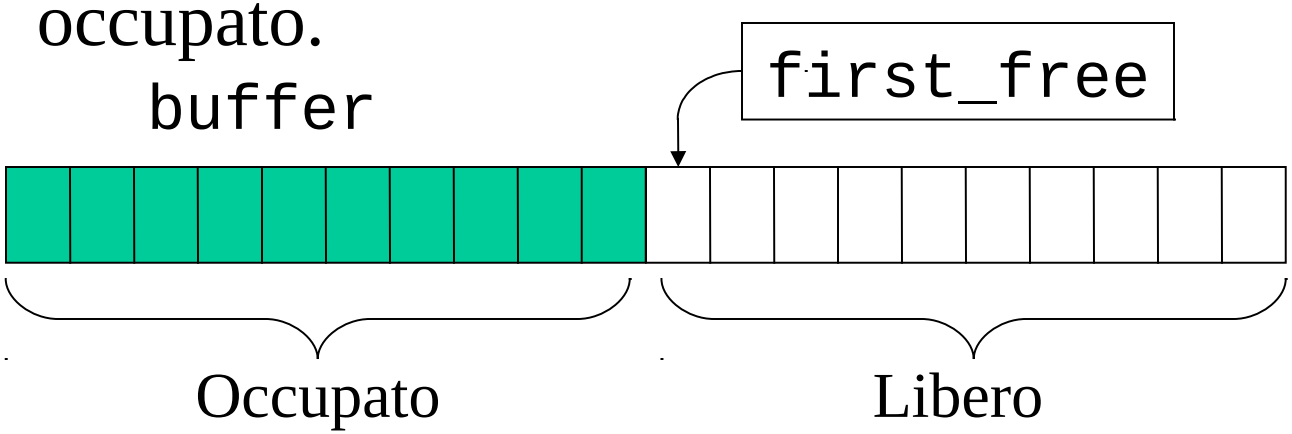
  `(*p)--` accede a `a [5]` e poi lo decrementa.

Pericoli nelle operazioni tra puntatori

- Se un puntatore va a puntare **al di fuori del vettore** (tranne che all'elemento immediatamente seguente) o se si effettuano operazioni illegali con puntatori, si hanno comportamenti imprevedibili.
  - Non esistono controlli né in compilazione né in esecuzione.

Esempio di aritmetica di puntatori (1)

- Gestione dinamica della memoria.
  - Si vuole gestire un buffer, parzialmente occupato.



  - Servono una funzione per allocare elementi e una per liberarli.

Esempio di aritmetica di puntatori (2)

`char* buf_alloc(unsigned int n)`

alloca `n` elementi, producendo il puntatore al primo.

`void buf_free(char* p)`

libera gli elementi a partire da quello puntato da `p`.

```
# define ALLOCSIZE 1000

char       buffer [ALLOCSIZE];
char*      first_free = buffer;
```

Esempio di aritmetica di puntatori (3)

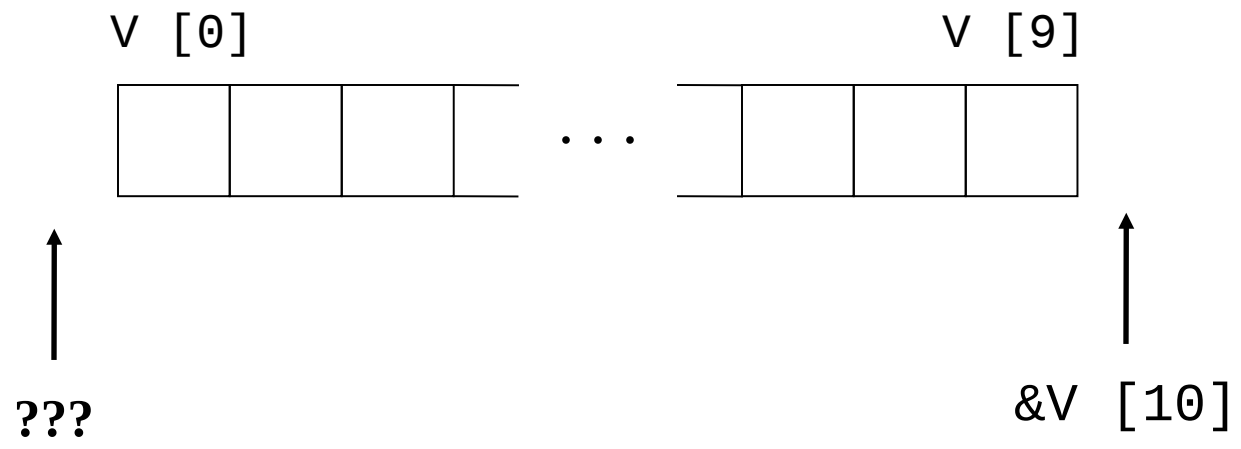
```
char*   buf_alloc(unsigned int n)
  {
  if (first_free > buffer +
     ALLOCSIZE - n)
     return NULL;
  first_free += n;
  return first_free - n;
  }
```

Esempio di aritmetica di puntatori (4)

```
void buf_free(char *p)
  {
  if (p < first_free)
     first_free = p;
  }
```

Attenzione ai puntatori (1)

```
int v [10], *p;
```



Se un puntatore va a puntare **prima** del primo elemento, diventa **indefinito**.

Se un puntatore va a puntare **una posizione oltre** l'ultimo elemento, resta **definito**.

Attenzione ai puntatori (2)

Sarebbe sbagliato scrivere:

```
if (firstfree + n >
      buffer + ALLOC_SIZE) …
```

perché se `n` è troppo grande,

```
firstfree + n
```

punta oltre la fine del vettore e il risultato del confronto è **indefinito**.

Attenzione ai puntatori (3)

L'istruzione:

```
for (p = v; p <= &v [9]; ++p) …
```

ha il comportamento atteso, ma

```
for (p = &v[9]; p >= v; --p) …
```

può comportarsi in modo inatteso, perché l'ultimo decremento porterà `p` a puntare a un inesistente elemento **prima** del vettore `v`, quindi a diventare **indefinito** e il confronto `p >= v` può dare risultati indefiniti.

- Su macchine a memoria segmentata, il secondo ciclo può essere infinito!

Forme alternative di ciclo

- Se è necessaria la scansione all'indietro, il ciclo va scritto in forma lievemente differente, evitando comunque di puntare all'elemento che **precede** il vettore.

Esempi di forme alternative di ciclo

```
for (p = &v[10]; p > v;)
   {
   --p;
   …
   }
```

oppure:

```
p = &v [9];
for (;;)
   {
   …
   if (p == v)
          break;
   --p;
   }
```

Forme di ciclo da evitare

- Va evitata la forma:

```
p = &v [9];
for (;;)
    {
    …
    if (p-- == v)
        break;
    }
```

perché, sebbene il confronto dia il risultato corretto, il decremento di `p` all'ultimo ciclo potrebbe provocare un errore.