Laboratorio di Programmazione

Corso di Laurea in Informatica

Gr. 3 (N-Z)

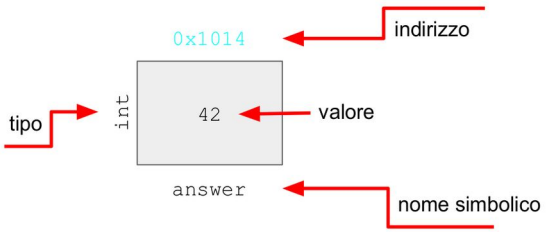
Università degli Studi di Napoli Federico II

A.A. 2022/23

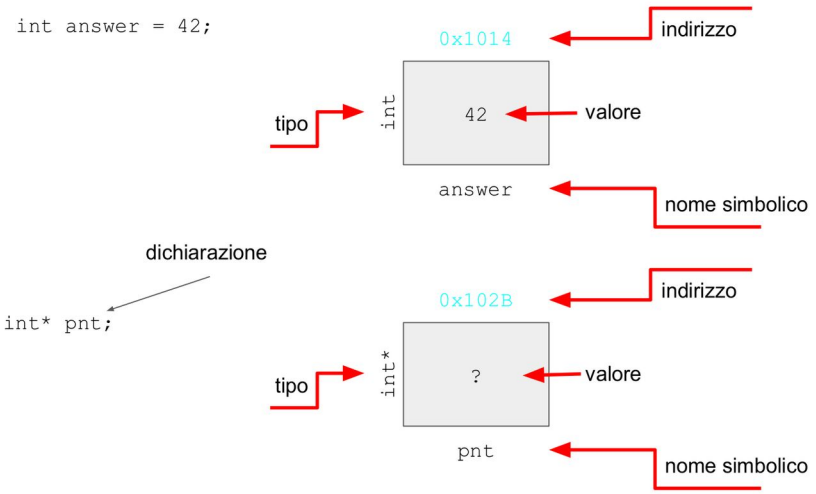
A. Apicella

Remembering C objects…

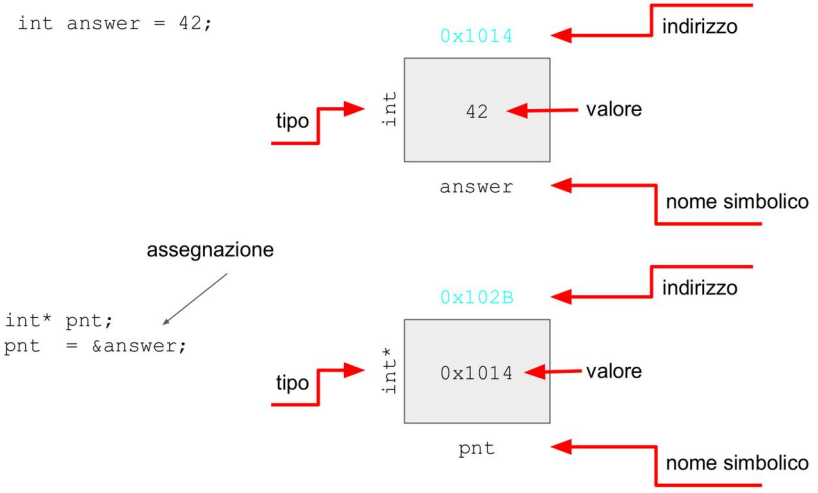
```
int answer = 42;
```



C pointers

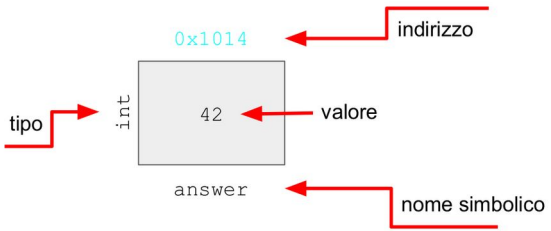


C pointers



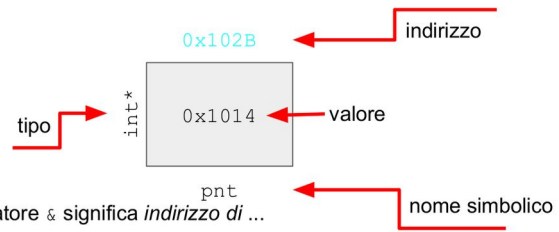
C pointers

```
int answer = 42;
```




```
int* pnt;
pnt  = &answer;
```

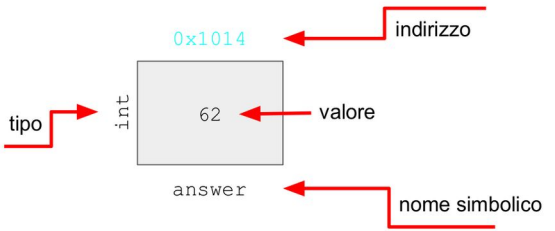
indirizzo dell'object answer



l'operatore `&` significa *indirizzo di ...*

C pointers

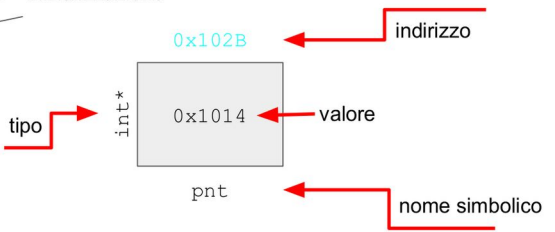
```
int answer = 42;
```



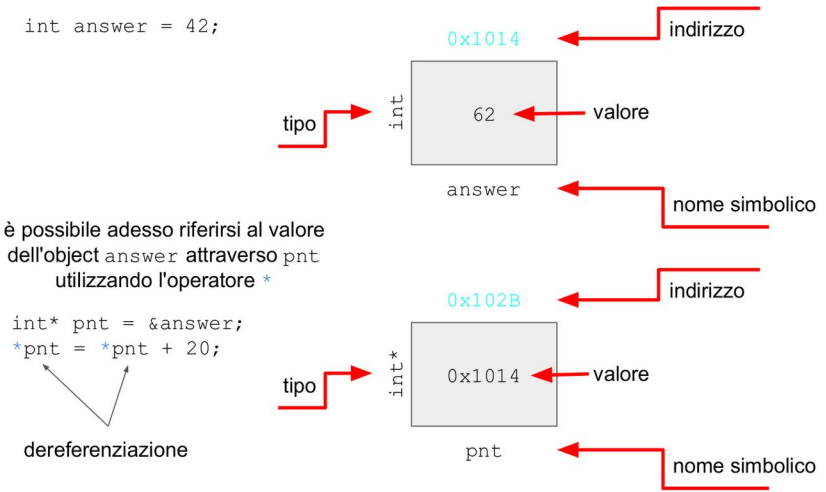
Equivalente a…

dichiarazione + inizializzazione

```
int* pnt = &answer;
*pnt     = *pnt + 20;
```



C pointers



C pointers

L'operatore *

l'operatore * può acquisire due significati differenti in base alla fase in cui viene utilizzato:

1. in fase di ***dichiarazione*** (e.g, `int* pnt;`): dichiara la variabile `pnt` di tipo *puntatore a int.* In questa fase, l'operatore * è **sempre** preceduto dal tipo dell'object di cui il puntatore può contenere l'indirizzo (in questo caso `int`), ed indica che si desidera dichiarare una variabile di tipo puntatore. Esempi:
   a. `int* pnt;`
   b. `float* pluto = &pippo;` (con `pippo` dichiarato come `float pippo;`)

   Ricorda che `float* pluto = &pippo;` significa:
   1. dichiara una variabile `pluto` di tipo `int*`
   2. inserisci al suo interno l'indirizzo dell'object `pippo`

2. in fase di ***dereferenziazione*:** con * si desidera accedere al valore dell'object il cui indirizzo è contenuto nel puntatore, dichiarato in precedenza. Si dirà quindi che il puntatore *punta* all'object. Esempi:
   a. `printf("%d",*pnt);`
   b. `int copy = *pnt;` (con `pnt` dichiarato come `int* pnt;`)

   Ricorda che `int copy = *pnt;` significa:
   1. dichiara una variabile `copy` di tipo `int`
   2. inizializza `copy` col valore a cui punta `pnt`

C pointers

L'operatore *

La posizione dell'operatore * in fase di dichiarazione è ininfluente, ossia:

`int* pnt;` ⇔ `int * pnt;` ⇔ `int *pnt;`

`int* pnt=&answer;` ⇔ `int * pnt=&answer;` ⇔ `int *pnt=&answer;`

Concentriamoci un attimo sulla sintassi più usata, ossia `int *pnt;`
Un altro modo di leggere `int *pnt;` è il seguente: dichiara una variabile di nome `pnt` tale che la combinazione `*punt` sia un `int`

ATTENZIONE:

`int *pnt=&answer;` ⇎ **`*pnt=&answer;`**

Non fatevi ingannare dal `*` attaccato al nome dell'object nel primo caso! Ricordatevi che siete in fase di dichiarazione (semplificando, quando alla sinistra di **`*`** c'è un tipo)

C pointers

L'operatore *

L'utilizzo della sintassi `type *pointer_name` ha come pro l'evitare ambiguità in caso di dichiarazioni multiple.

Esempio:

`int* p,q;`
dichiara la variabile `p` di tipo puntatore a `int`, mentre la variabile `q` sarà di tipo int ma non puntatore.
In sostanza, l'operatore `*` viene applicato solo al primo object. e non agli altri.

Per avere entrambe le variabili di tipo puntatore a `int`:

- `int* p,*q;` oppure
- `int * p, * q;` oppure...
- `int* p;`
  `int* q;`

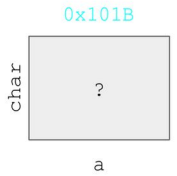
C pointers

L'operatore &

L'operatore `&` restituisce l'indirizzo di memoria dell'object a cui è applicato

```
char a;
printf("%d", &a);
```

Stamperà la codifica decimale dell'indirizzo `0x101B` dell'object `a`



`%p` è uno specificatore di formato creato appositamente per stampare indirizzi di memoria

```
printf("%p", &a);
```

Il formato effettivo di stampa dipende dall'implementazione della funzione `printf` (solitamente viene stampato il valore in esadecimale)

ATTENZIONE:
Potreste trovare del codice con istruzioni del tipo:

`int& r = q;` oppure `int &r = q;` oppure `int&r = q;`

Se trovate una istruzione del genere, allora non state avendo a che fare con il linguaggio *C*, ma con il linguaggio C**++**.
In C**++** l'operatore &, se posto di fianco ad un tipo in fase di dichiarazione non è più operatore *indirizzo,* ma di *aliasing*.
L' *aliasing* non esiste in C! E' una caratteristica del C++, e come tale non è oggetto di questo corso!

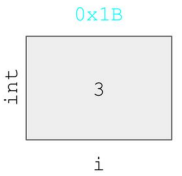
C pointers

void pointers

Un puntatore di tipo `void` (ossia `void*`) può contenere l'indirizzo di un object di qualsiasi tipo

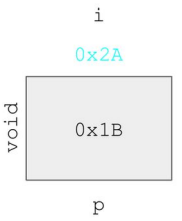
```
void* p;
int i = 3;
p     = &i;
```

nessun problema. Ma…




```
printf("%d",
*p);
```

error: invalid use of void expression



La dereferenziazione di un puntatore `void` in generale non ha senso.

- Il compilatore deve sapere *di che tipo è* la memoria a cui sta accedendo, così da poterla trattare nella maniera corretta (e.g. il valore da gestire da quanti byte è composto? devo considerarlo con segno o no? ecc.)
- necessario un **cast** affinché la dereferenziazione sia effettuata in maniera corretta.

```
printf("%d", *( (int*)p));
```

C pointers

Buone regole di puntamento

- In C, la sola dichiarazione di un puntatore non assegna automaticamente un valore!

  ⇒ Evitare di dichiarare puntatori senza inizializzarli!

  `int* p;`

  In tal caso `p`, se allocato nel call stack, potrebbe contenere al suo interno qualsiasi valore. Una sua eventuale deferenziazione `*p = ...` per assegnargli un valore può essere pericolosa!

  Esempio:

  - Si supponga che casualmente `p` contiene l'indirizzo di memoria di una object `const`…

  Generalmente, molti bugs nascono da una cattiva/mancata inizializzazione dei puntatori.

- Se non si conosce ancora quale sarà l'object a cui dovrà puntare `p` o non è stato ancora dichiarato, meglio inizializzarlo con `NULL`.

  `int* p = NULL;`

  `NULL` è una macro definita nell' header *stdio.h* corrispondente all'indirizzo 0.

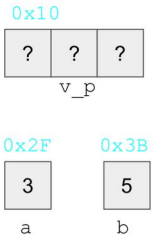
  In C è garantito che nessun object esiste né esisterà all'indirizzo NULL. Attraverso questa convenzione, è possibile controllare se un puntatore punta o meno ad un oggetto (e.g., `if(p == NULL){...}`).
- Dereferenziare un puntatore che (non) punta a null(a) è concettualmente sbagliato, anche se sintatticamente corretto

```
int* p = 0;
*p     = 10; //sbagliato! ma nessun errore da parte del compilatore
```

C pointers

Arrays of pointers

```
int* v_p[3];
int a=3, b=5;
```

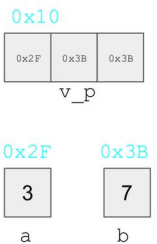


C pointers

Arrays of pointers

```
int* v_p[3];
int a=3, b=5;
v_p[0] = &a;
v_p[1] = &b;
v_p[2] = &b;

*v_p[1] = 7;
```

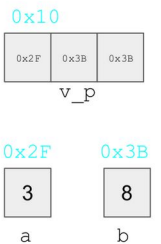


C pointers

Arrays of pointers


```
int* v_p[3];
int a=3, b=5;
v_p[0] = &a;
v_p[1] = &b;
v_p[2] = &b;

*v_p[1] = 7;
*v_p[2] = 8;
```

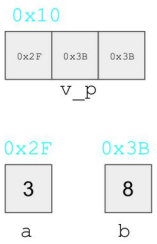


C pointers

Arrays of pointers

```
int* v_p[3];
int a=3, b=5;
v_p[0] = &a;
v_p[1] = &b;
v_p[2] = &b;

*v_p[1] = 7;
*v_p[2] = 8;
 v_p[0] = 9;
```

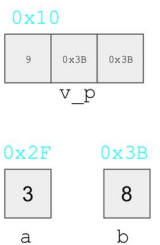


C pointers

Arrays of pointers

```
int* v_p[3];
int a=3, b=5;
v_p[0] = &a;
v_p[1] = &b;
v_p[2] = &b;

*v_p[1] = 7;
*v_p[2] = 8;
 v_p[0] = 9;
```

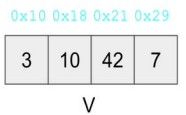
C pointers

Arrays as pointers (almost always)

Except when it is the operand of the `sizeof` operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.
C standard, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

Il nome di un array può essere utilizzato, nella maggioranza delle volte, come se fosse un puntatore costante (ossia il cui indirizzo memorizzato non può essere modificato). Tale "puntatore" punta al primo elemento dell'array

```
int v[4] = {3, 10, 42, 7};
printf("indirizzo usando v: %p", v);
printf("indirizzo usando &v[0]: %p\n", &v[0]);
printf("1° valore di v visto come array: %d\n", v[0]);
printf("1° valore di vettore v visto come puntatore: %d\n", *v);
```



Output:
indirizzo usando v: 0x10; indirizzo usando &v[0]: 0x10
primo valore del vettore v visto come array: 3
primo valore del vettore v visto come puntatore: 3

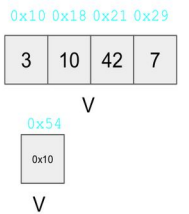
C pointers

Arrays as pointers (almost always)

Except when it is the operand of the `sizeof` operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.
C standard, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

Eccezione: operatore `sizeof`

```
int V[4] = {3, 10, 42, 7};
int *p_V = V;
printf("V: %p; p_V: %p; &V: %p; &p_V: %p\n", V, p_V, &V, &p_V);
printf("sizeof(V): %d Byte\n", sizeof(V));
printf("sizeof(p_V): %d Byte\n", sizeof(p_V));
```



Output:
V: 0x10; p_V: 0x10; &V: 0x10; &p_V: 0x54
sizeof(V): 12 Byte
sizeof(p_V): 8 Byte

pur puntando allo stesso vettore, l'operatore `sizeof` restituisce:
- la dimensione effettiva dell'array se applicato al nome dell'array
- la dimensione del puntatore se applicato ad un puntatore reale

C pointers

Arrays in functions' parameters

Se un array viene passato come parametro ad una funzione, l'array viene trattato esattamente come un puntatore, *indipendentemente se questo è passato come array o come puntatore*.

`void f(int V[])` ⇔ `void f(int* V)`

⇒ quando il nome di un array passato come parametro di una funzione, un array non è mai passato per copia, ma per indirizzo (in quanto il nome corrisponde all'indirizzo del primo elemento)

⇒ con array passati come parametri, l'operatore `sizeof` si comporta come con i puntatori

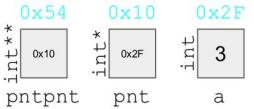
```
void f(int f_v[]) // equivalente a void f(int* f_v)
{
     printf("sizeof(f_v): %d Byte\n", sizeof(f_v));
}
int main()
{
     int v[] = {3, 10, 42};
     int *p_v = v;
     printf("sizeof(v):   %d Byte\n", sizeof(v));
     printf("sizeof(p_v): %d Byte\n", sizeof(p_v));
     f(v);
     return 0;
}
```

Output:
sizeof(v): 12 Byte
sizeof(p_v): 8 Byte
sizeof(f_v): 8 Byte

C pointers

Pointers to pointers

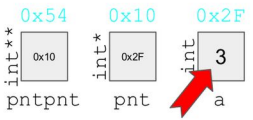
```
int  a      = 3;
int* pnt    = &a;
int** pntpnt = &pnt;
```



C pointers

Pointers to pointers

```
int  a       = 3;
int* pnt     = &a;
int** pntpnt = &pnt;
```




```
printf("%d", **pntpnt)
```

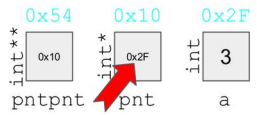
oppure…

```
printf("%d", a)
```


C pointers

Pointers to pointers

```
int  a       = 3;
int* pnt     = &a;
int** pntpnt = &pnt;
```


```
printf("%d", *pntpnt)
```

oppure…

```
printf("%d", pnt)
```

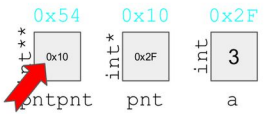
oppure…

```
printf("%d", &a)
```

C pointers

Pointers to pointers

```
int  a       = 3;
int* pnt     = &a;
int** pntpnt = &pnt;
```

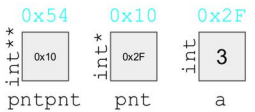



```
printf("%d", pntpnt)
        oppure…
printf("%d",  &pnt)
```


C pointers

Pointers to pointers


```
int  a       = 3;
int* pnt     = &a;
int** pntpnt = &pnt;
```



possono essere dichiarati puntatori con qualsiasi livello di puntamento, i.e.
`int*`, `int**`, `int***`, `int****` e così via

C pointers

Bidimensional arrays as pointers to pointers (?)

Dato un array bidimensionali tipo

```
int M[4][2];
```

`M` può essere considerato (quasi) equivalente ad un puntatore a puntatore

```
int** M
```

?

NO!

In generale, un array multidimensionale è visto come un puntatore *a elementi di tipo array*, ossia

```
int (*) [ ]
```

un tipo particolare di puntatore.

```
int (*) [ ] ≠ int**
```

i puntatori a elementi di tipo array non saranno argomento di questo corso.