5.1 – Gestione della memoria e puntatori

Libro di testo:

- Capitolo 17.3, 17.3.1

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Gestione della memoria
- Puntatori e indirizzi
- Accesso ai dati puntati
- Tipi di puntatori
- `sizeof`

Gestione della memoria

- Un compito di basso livello
  - Spesso demandato alle librerie
- Un compito utile da studiare
  - Imparare a usare la memoria per capire come lo fanno le librerie
  - Non "credere nella magia"
  - Essere in grado di implementare nuovi container

Motivazione

"More philosophically, I am among the large group of computer professionals who are of the opinion that if you lack a basic and practical understanding of how a program maps onto a computer's memory and operations, you will have problems getting a solid grasp of higher-level topics, such as data structures, algorithms, and operating systems" (BS)

Puntatori

- È un particolare tipo di variabile
- Un puntatore contiene **un indirizzo di memoria**
  - Così come un `int` contiene un intero, un `bool` un valore logico, …

`ptr`: `0x1001054a0`

- Spesso l'indirizzo di memoria non è importante in sé, ma perché è possibile usarlo per accedere al dato puntato

Definizione di un puntatore

- Un puntatore punta a un tipo
  - È noto il tipo di dato che si trova all'indirizzo di memoria contenuto nel puntatore
- La definizione contiene
  - Il tipo puntato
  - Il carattere * che significa che la variabile definita è un puntatore

```
double d;   // variabile double

double* p;  // variabile puntatore a double
```

Definizione di un puntatore

- Il simbolo * è sibillino
  - Può essere unito al tipo o al puntatore
  - Può essere separato

```
double* p1;
double *p2;
double * p3;
```

Equivalenti

Definizione di un puntatore

- Per dichiarare più di un puntatore in una sola istruzione è necessario ripetere *

```
double *p1, *p2;   // due puntatori a double
double *p3, d1;    // un puntatore a double e un double
double* p4, d2;    // un puntatore a double e un double
double * p5, d3;   // un puntatore a double e un double
```

- La spaziatura non influisce sulla semantica
  - Ma può essere controintuitiva (es., terza riga)

Organizzazione della memoria

- La memoria è indirizzata a byte
  - Un indirizzo indica una locazione di memoria

- Questo è vero anche per ciò che abbiamo usato finora

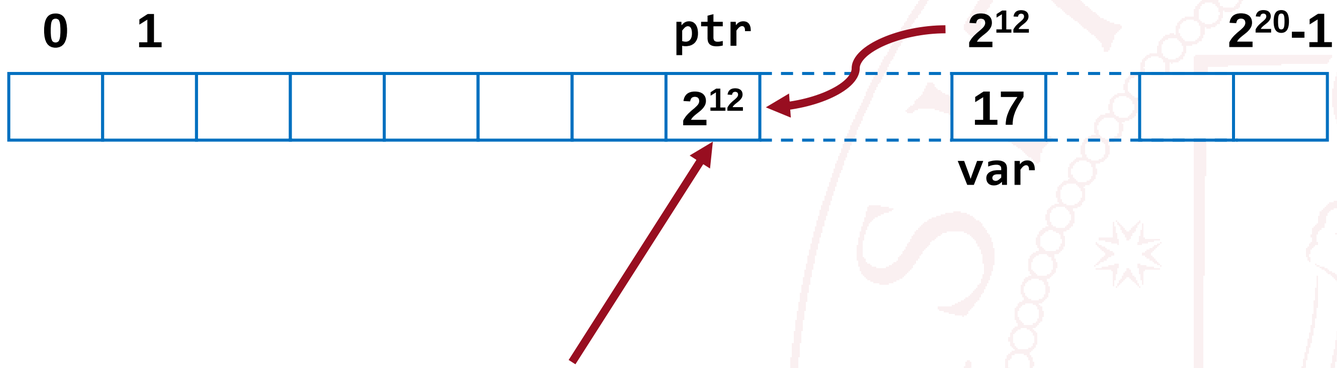
```
int var = 17;
```

  - Dov'è `var`? Quanti byte occupa?

Indirizzo di una variabile

- Supponiamo che `var` sia all'indirizzo 2^{12}
- È possibile acquisire l'indirizzo di una variabile con:

```
int* ptr = &var;
```



`ptr` è a tutti gli effetti una nuova variabile di tipo puntatore (`int`) immagazzinata in un'altra posizione nella memoria

Tipi di puntatori

- Ogni tipo ha il suo puntatore

```
int x = 17;
int* pi = &x;

double e = 2.71828;
double* pd = &e;
```

- Un puntatore ha un contenuto esprimibile con un numero intero (l'indirizzo di memoria), ma è un tipo a sé
  - Tale tipo supporta operazioni tra indirizzi

Accedere al dato puntato

- Vedere il contenuto della memoria puntata
  - Operatore dereference: *

```
int x = 17;
int* pi = &x;

double e = 2.71828;
double* pd = &e;
```

```
cout << "pi == " << pi << ", content of pi == " << *pi << '\n’;

cout << "pd == " << pd << ", content of pd == " << *pd << '\n';
```

Esempi

- L'operatore * di deference restituisce una lvalue:
  - Cosa significa?

```
int x = 17;
int* pi = &x;
double e = 2.71828;
double* pd = &e;

*pi = 27;           // ok
*pd = 3.14159;      // ok
*pd = *pi;          // cosa fa questo?
```

- Si può assegnare un `int (*pd)` a un `double (*pd)`
- Tipe check!

```
int i = pi;     // errore: tipi incompatibili (int* -> int)
pi = 7;         // errore (int -> int*)
```

Tipi di puntatori

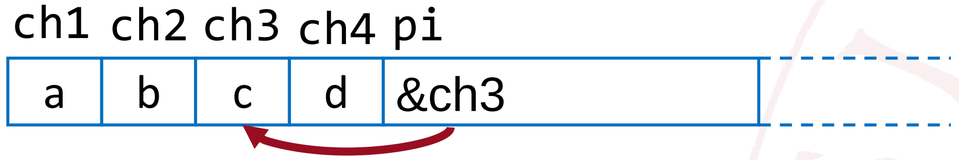
- Due puntatori a tipi diversi sono essi stessi tipi diversi

```
int x = 17;
int* pi = &x;

char* pc = pi;     // errore, non si può assegnare int* a char*
pi = pc;           // errore, non si può assegnare char* a int*
```

- Ciò è necessario perché è diverso ciò che posso fare col dato puntato

Tipi di puntatori

```
char ch1 = 'a';
char ch2 = 'b';
char ch3 = 'c';
char ch4 = 'd';

int* pi = &ch3;    // anche se potessi, cosa accadrebbe
                   // nelle righe seguenti?
*pi = 12345;
*pi = 67890;
```



- Con `*pi = 12345` si cambiano i valori anche di `ch2` e `ch4` Perchè?
- Nel caso più sfortunato andremo a sovrascrivere anche `pi`
- Nel assegnamento successivo `*pi=67890`, il valore viene scritto in una locazione non definite della memoria

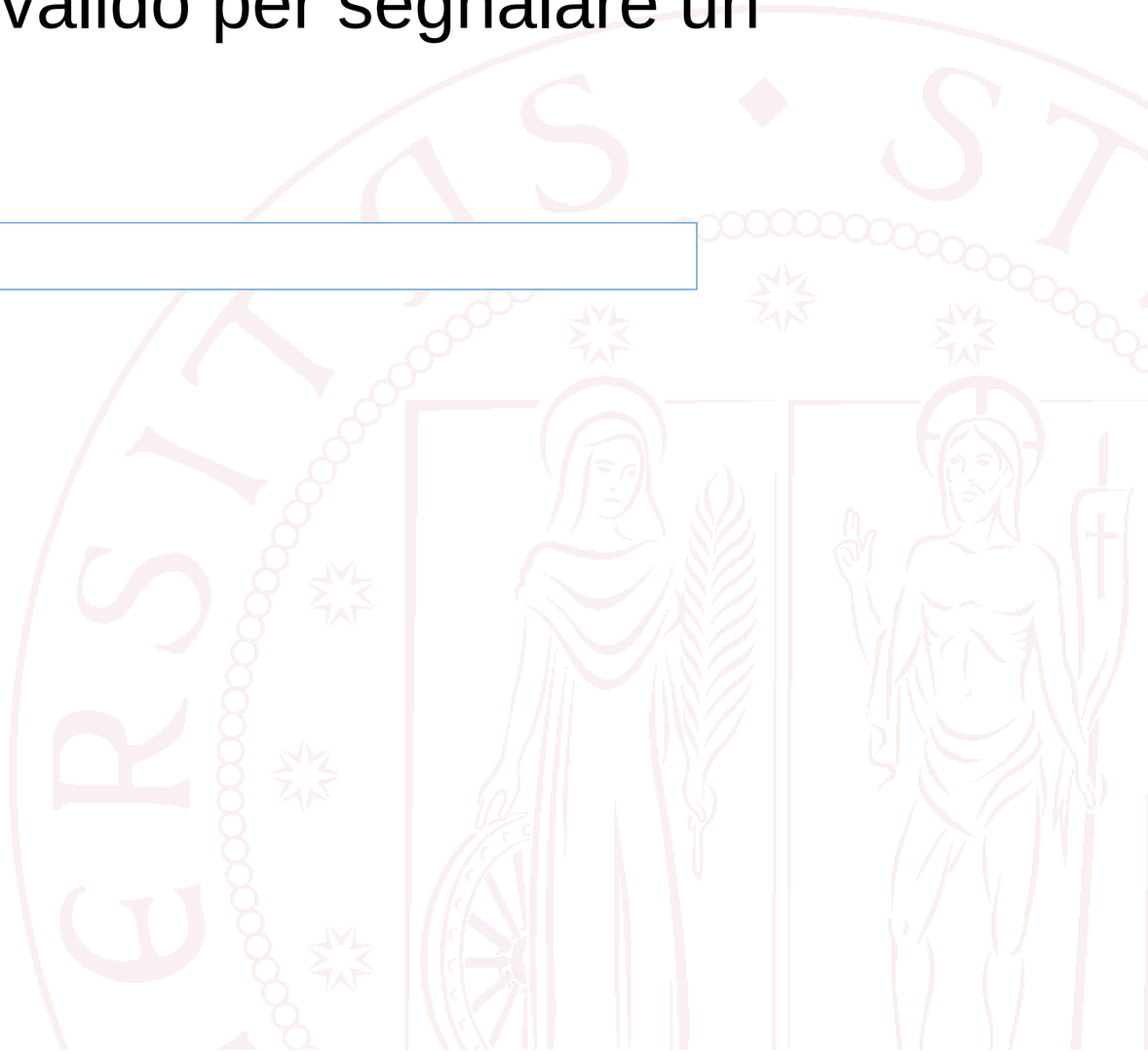
Per fortuna, `int* pi=&ch3;` non è legale

null pointer

- È molto comodo avere un valore non valido per segnalare un puntatore non inizializzato

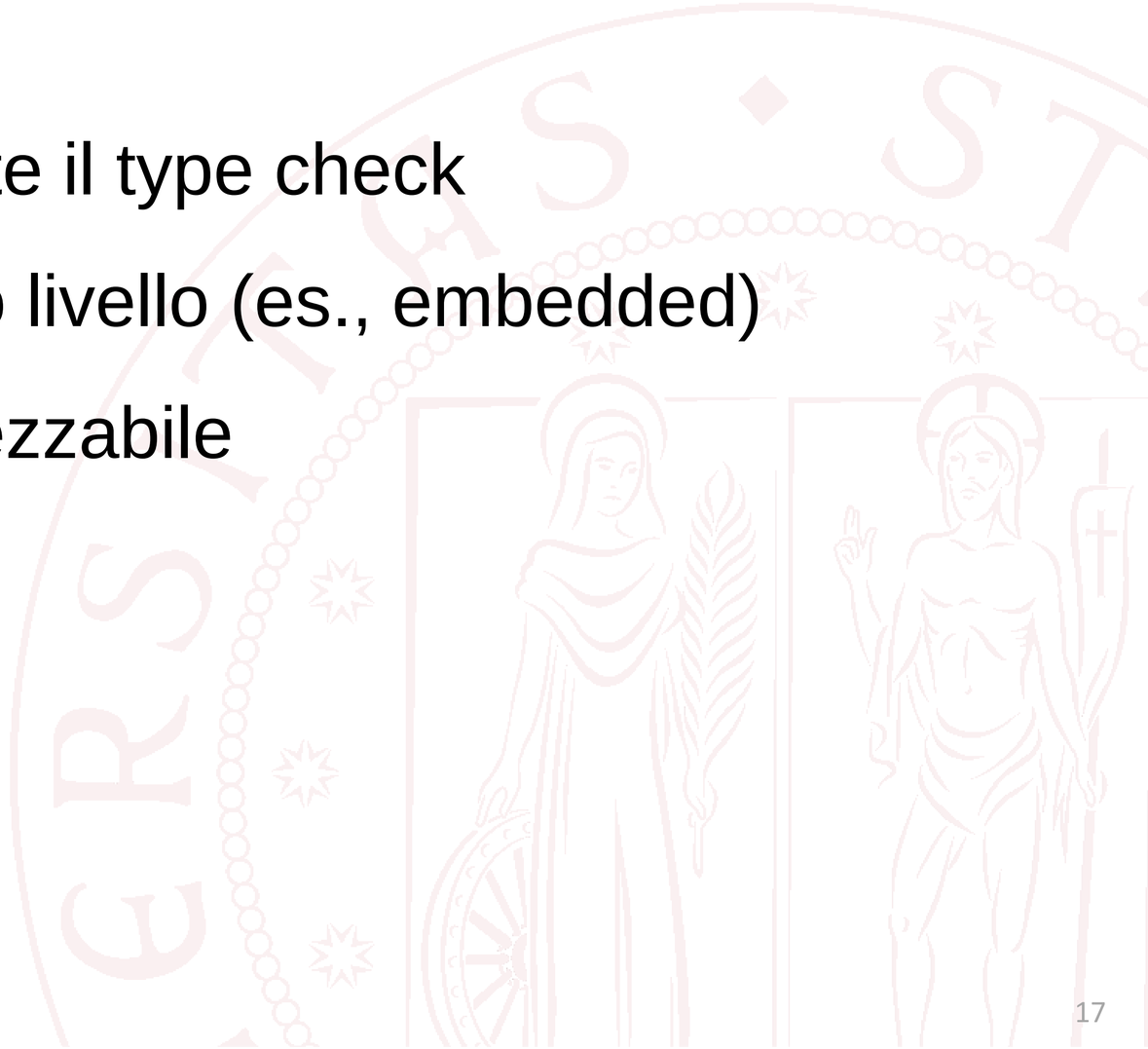
```
double* p0 = nullptr;
```

- `nullptr` è C++11
  - Altrimenti: `NULL` oppure `0`



Note sui puntatori

- Siamo molto vicini all'HW
  - Poco supporto, ma almeno è presente il type check
  - Non tutto il codice può essere ad alto livello (es., embedded)
  - Rende `std::vector` molto più apprezzabile



sizeof()

- In questo contesto è molto utile `sizeof()`
  - Funziona con un nome di tipo o di variabile/espressione

```
void sizes(char ch, int i, int* p) {
    cout << "The size of char is " << sizeof(char) << ' ' << sizeof(ch) << '\n';

    cout << "The size of int is "  <<  sizeof(int) << ' ' <<  sizeof(i) << '\n';

    cout << "The size of int* is " << sizeof(int*) << ' ' <<  sizeof(p) << '\n';

}
```

- Attenzione: la dimensione dipende dall'implementazione

Recap

- Organizzazione della memoria
- Concetto di puntatore
- Uso di un puntatore
  - Accesso al dato puntato
  - Legame tra puntatore e tipo puntato
- `sizeof`