

Richiami di C e C++

Il diagramma di *Memory Layout*

La notazione *Data Object*

Richiami di C: dichiarazioni e definizioni

```
struct Person {  
    char name[100];  
    int age;  
};
```

Dichiarazione di un nuovo tipo di dati

```
struct Person p;
```

Definizione di variabile di tipo Person, non
inizializzata

Se globale, viene automaticamente
inizializzata a zero (tutti zeri, anche i char)

```
struct Person {  
    char name[100];  
    int age;  
} p;
```

Dichiarazione del tipo e **definizione** della
variabile, congiunte

Inizializzazione di struct

```
struct Person {  
    char name[100];  
    int age;  
};
```

```
struct Person p1 =  
    { "Pippo", 30 };
```

Definizione e inizializzazione (vecchio stile)

```
struct Person p2 = {  
    .name = "Pippo",  
    .age = 30  
};
```

Definizione e inizializzazione (nuovo stile C99)

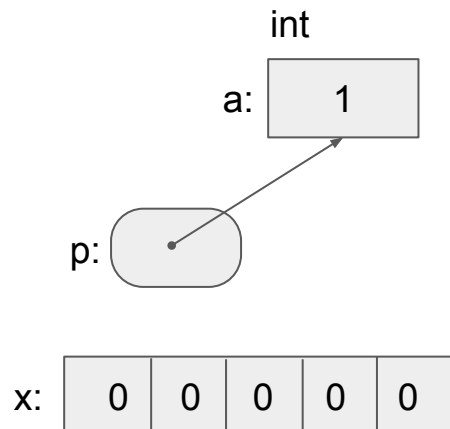
Diagramma di Memory Layout

- Mostra le allocazioni di memoria contigue
- Il valore di puntatori e riferimenti è rappresentato con frecce
- Non distingue tra stack e heap
- Versione grafica di *data object*

Codice C:

```
int a = 1  
int *p = &a;  
int x[5] = { 0, };
```

Memory layout:



Data Objects:

(loc1, a, 1, int)
(loc2, p, loc1, puntatore a int)
(loc3, x, {0,0,0,0,0}, array di int)

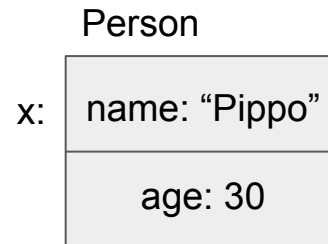
Memory Layout di struct (1)

Codice C:

```
struct Person {  
    char name[100];  
    int age;  
} x = {  
    .name = "Pippo",  
    .age = 30  
};
```

`sizeof(Person)` → 104

Memory layout:



Data Objects:

principale:
(loc1, x, {"Pippo", 30}, struct Person)

aggiuntivi:
(loc1, x.name, "Pippo", char[100])
(loc1+100, x.age, 30, int)

Memory Layout di struct (2)

Codice C:

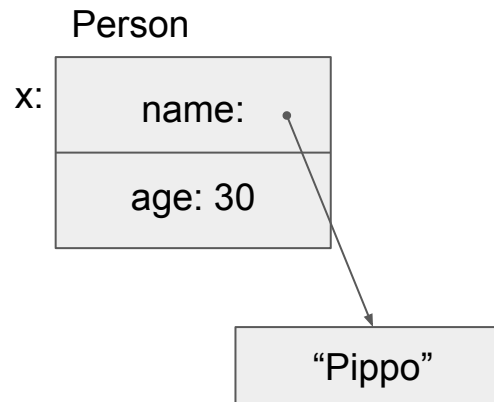
```
struct Person {  
    char *name;  
    int age;  
} x = {  
    .name = "Pippo",  
    .age = 30  
};
```

`sizeof(Person)` → 16

Perché 16?

alignment e padding

Memory layout:



Data Objects:

principali:

(loc1, x, {loc2, 30}, struct Person)

(loc2, anonimo, "Pippo", const char[])

aggiuntivi:

(loc1, x.name, loc2, char*)

(loc1+8, x.age, 30, int)

ipotizzando che `sizeof(char*)` sia 8

Esempio in Java: interi e “Interi”

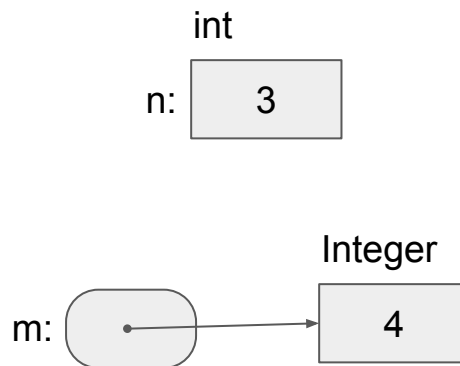
Codice Java:

```
int n = 3;
```

```
Integer m = 4;
```

Nota: grazie all'*autoboxing*, la seconda istruzione è equivalente a
`Integer m = Integer.valueOf(4)`

Memory layout:



Data Objects:

(loc1, n, 3, int)

(loc2, *anonimo*, 4, Integer)

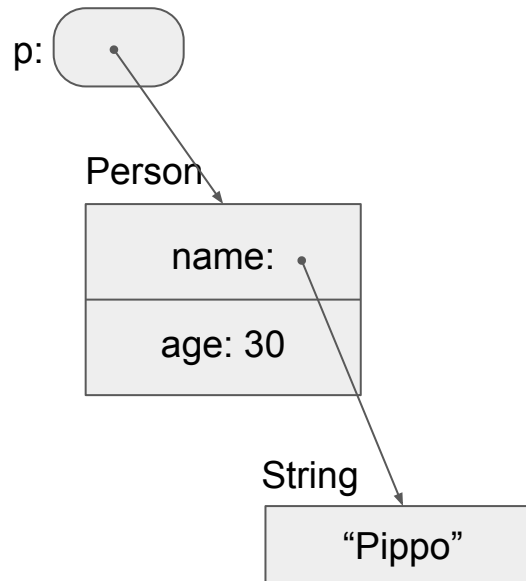
(loc3, m, loc2, “riferimento a Integer”)

Esempio in Java: classi

Codice Java:

```
class Person {  
    String name;  
    int age;  
    ...  
}  
  
Person p = new  
Person("Pippo", 30);
```

Memory layout:



Data Objects:

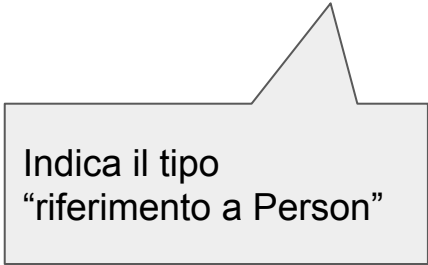
(loc1, *anonimo*, ..., Person)
(loc2, p, loc1, "riferimento a Person")
(loc3, *anonimo*, "Pippo", String)

Nota: questo memory layout è una semplificazione.
In realtà la stringa "Pippo" si trova in un array di
caratteri che è separato dall'oggetto di tipo String.

Tipi riferimento in Java

Codice Java:

```
Person p = new Person("Pippo", 30);
```



Indica il tipo
“referimento a Person”



Indica il tipo Person

- Quando il nome di un tipo T (non primitivo) viene usato per creare un oggetto (cioè, new T(...)), crea un oggetto della classe T
- Quando il nome di un tipo T (non primitivo) viene usato per dichiarare una nuova variabile, indica il tipo “referimento a T”

Richiami di C++: classi e allocazione automatica

```
class Person {  
private:  
    string name;  
    int age;  
public:  
    Person(string n, int a): name(n), age(a) { }  
};
```

costruttore

sezione di
inizializzazione

corpo vuoto

Allocazione automatica di un oggetto Person:
(su stack se locale, su segmento statico se globale)

```
Person p = Person("Pippo", 30);
```

oppure:

```
Person p("Pippo", 30);
```

oppure:

```
Person p { "Pippo", 30 };
```

Richiami di C++: riferimenti

- Un riferimento è un alias, implementato come **puntatore costante**
- Sintatticamente, non si comporta come un puntatore, ma come l'oggetto puntato
- Uso tipico: per realizzare il passaggio di parametri per riferimento
 - Se il metodo non deve modificare l'argomento, si usa `const T&`
 - Se il metodo modifica l'argomento, si usa `T&`

Con puntatore:

```
void f(struct Big *big) {  
    cout << big->field;  
    cout << big;    ⇨ Indirizzo della struct  
    cout << *big;    ⇨ Contenuto della struct  
    big->field = 0;  
    big = nullptr;  
}
```

Con riferimento:

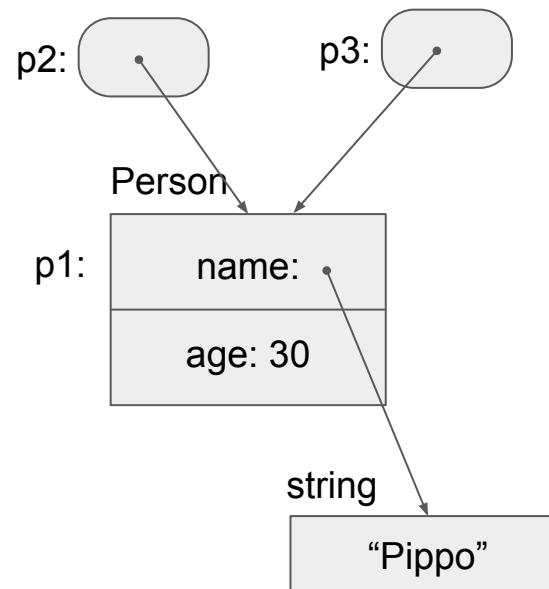
```
void f(struct Big &big) {  
    cout << big.field;  
    cout << big;    ⇨ Contenuto della struct  
    cout << &big;    ⇨ Indirizzo della struct  
    big.field = 0;  
    big = nullptr;    ⇨ Errore di comp.  
}
```

Esempio in C++:

Codice C++:

```
class Person {  
    string name;  
    int age;  
    ...  
};  
  
Person p1 =  
Person("Pippo", 30);  
  
Person &p2 = p1;  
Person *p3 = &p1;
```

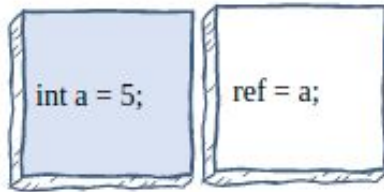
Memory layout:



Data Objects:

(loc1, p1, ..., Person)
(loc2, p2, loc1, riferimento a Person)
(loc3, p3, loc1, puntatore a Person)
(loc4, *anonimo*, "Pippo", string)

Attenzione! (al materiale che si trova online)



Creating a reference to **a** just makes an alias for it;
it does not "point" to **a** by storing its address in a
separate memory location

Da:

[https://www.educative.io/edpresso/
differences-between-pointers-and-r
eferences-in-cpp](https://www.educative.io/edpresso/differences-between-pointers-and-references-in-cpp)

Errato!

I riferimenti *si comportano* come “alias”, ma
sono implementati come puntatori costanti

Dimostrazione pratica:

```
class Person { ... }
```

```
struct PersonRef {  
    Person &p;  
}
```

```
sizeof(Person) → 40
```

```
sizeof(PersonRef) → 8
```

Data Model a confronto

Java

Le variabili non primitive contengono riferimenti

Le variabili primitive contengono valori

C#

Classi e array si manipolano per riferimento

I tipi primitivi e le struct per valore

C++

Per qualsiasi tipo, si può scegliere tra valore, puntatore e riferimento