



Università degli Studi di Cassino

Corso di Fondamenti di Informatica

Puntatori

Anno Accademico 2011/2012

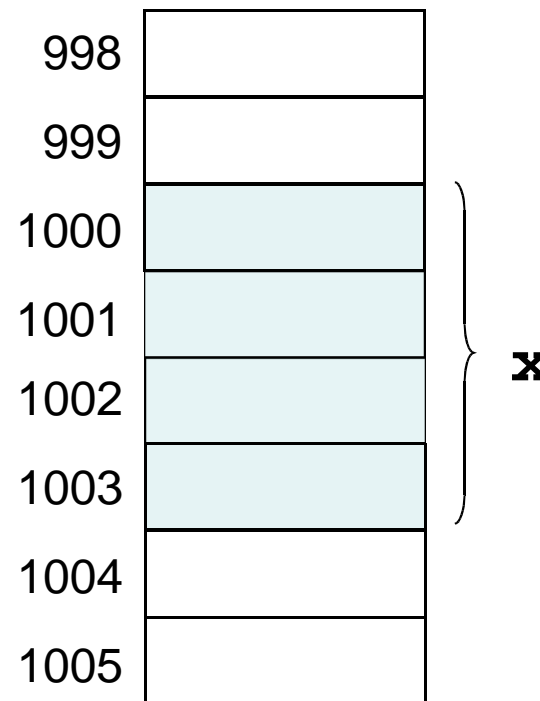
Francesco Tortorella

Variabili, registri ed indirizzi

- Abbiamo visto che la definizione di una variabile implica l'allocazione (da parte del compilatore) di registri di memoria. Il numero di registri allocati dipende dal tipo della variabile.
- Alla porzione di memoria allocata si accede tramite l'identificatore della variabile. Questo ci risparmia di preoccuparci in quale particolare locazione la variabile sia realmente allocata.
- È il compilatore a creare e gestire la corrispondenza tra identificatore della variabile e indirizzo della locazione in memoria.

Variabili, registri ed indirizzi

- Esempio:
`int x;`
- Con l'istruzione viene definita una variabile intera **x** che occupa 4 registri da 1 byte a partire dall'indirizzo 1000.



Variabili, registri ed indirizzi

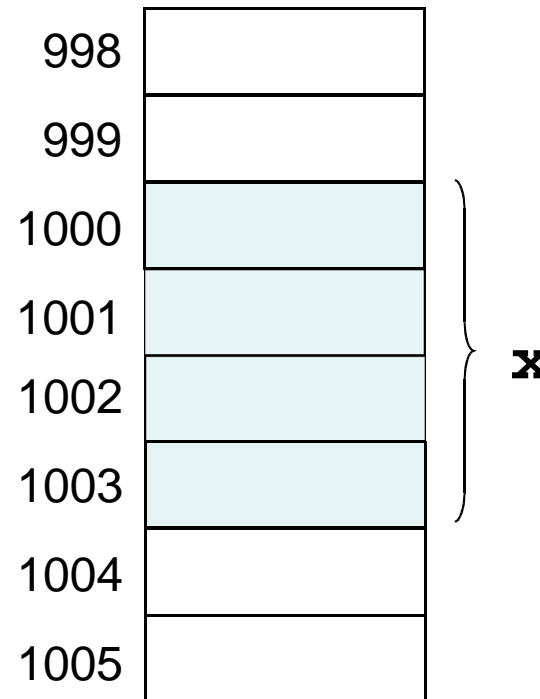
- Di fatto, abbiamo considerato l'indirizzo di una variabile quando abbiamo descritto il passaggio di parametri per riferimento.
- In quel caso il compilatore accede ed utilizza l'indirizzo della variabile che viene passato al sottoprogramma invocato.
- Il C++ dà la possibilità di accedere esplicitamente all'indirizzo di una variabile tramite l'operatore & (operatore di riferimento o di *reference*) prefisso all'identificatore della variabile.

`int x;` variabile `x`

`&x` indirizzo della variabile `x`

Variabili, registri ed indirizzi

- Esempio:
`int x;`
- In questo caso `&x` sarà uguale a 1000.
- **ACHTUNG !**
L'operatore `&` si può applicare solo alle variabili (o, più precisamente, a *l-value*)



Variabili, registri ed indirizzi

```
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    int x=3;

    cout << "Valore di x: " << x << endl;
    cout << "Indirizzo di x: " << &x << endl;
    return (EXIT_SUCCESS);
}
```

Valore di x: 3

Indirizzo di x: 0x22ff48 ← indirizzo esadecimale

Puntatori

- Il C++ permette di definire delle variabili di tipo “puntatore” cui si possono assegnare gli indirizzi di variabili di un particolare tipo.
- La definizione di tali variabili (dette **puntatori**) richiede la specificazione del tipo “puntato”, seguito da un ‘*’.
- Es.: definizione di un puntatore a `int`
`int *p;`

Puntatori

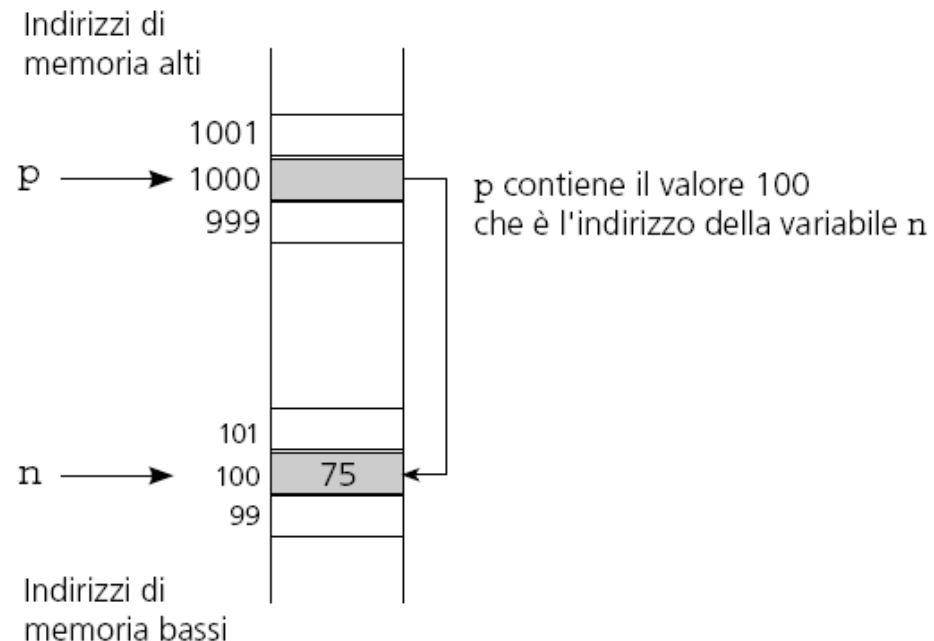
Di fatto una variabile di tipo *puntatore al tipo T* contiene l'indirizzo di memoria di una variabile di *tipo T*.

Esempio:

```
int n;  
int *p;
```

```
n = 75;
```

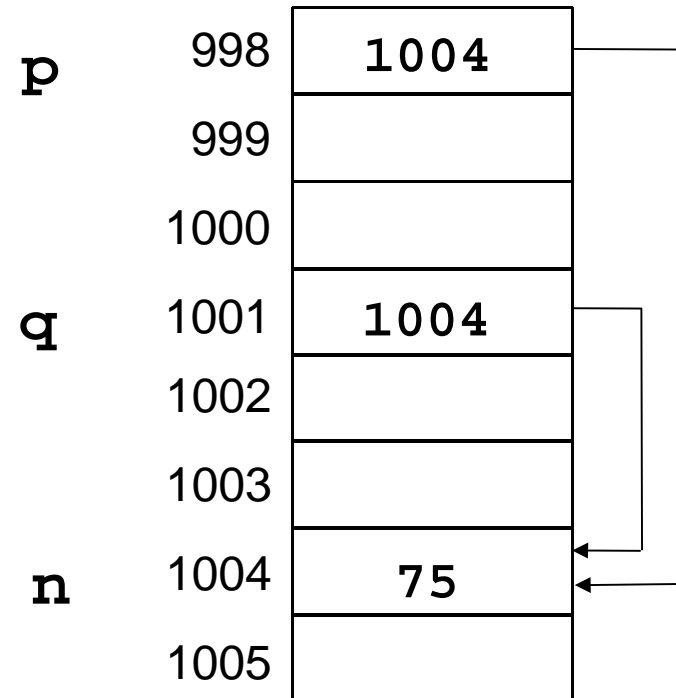
```
p = &n;
```



Puntatori

- È possibile fare assegnazioni tra puntatori.
- Esempio:

```
int n=75;  
int *p,*q;  
p = &n;  
q = p;
```
- In questo modo due puntatori puntano alla stessa variabile.



Puntatori

- Tramite il puntatore è possibile accedere alla variabile puntata.
- Con l'operatore * (operatore di indirizione o di *dereference*) prefisso all'identificatore della variabile puntatore è possibile accedere direttamente alla variabile puntata, sia in lettura che in scrittura.
- In questo modo si crea un *alias* della variabile che può essere modificata tramite il puntatore.

Puntatori

```
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    int x=3,y=5;
    int *p;

    p = &x;
    *p = 10;
    cout << "Valore di x: " << x << endl;

    p = &y;
    *p = 20;
    cout << "Valore di y: " << y << endl;

    return (EXIT_SUCCESS);
}
```

Valore di x: 10 Valore di y: 20

Puntatori

- Non è possibile fare assegnazioni tra puntatori di tipo diverso.

Es.:

```
int x=3, *p;  
char c='A', *t=&c;  
p=t; non è consentito
```

- Perché ? In fondo entrambi contengono un indirizzo.
- Domanda fondamentale: perché i puntatori si riferiscono a un tipo particolare ?

Puntatori

- Di fatto, p e t sono entrambi puntatori ed occupano lo stesso spazio in memoria (quanto?).
- Bisogna però ricordare che c'è una profonda differenza tra i dati puntati :
 - Non sono dello stesso tipo
 - Non occupano lo stesso spazio in memoria
 - Non utilizzano la stessa rappresentazione dei dati
- Per accedere correttamente alla variabile puntata è quindi necessario che il puntatore “porti con sé” l'informazione sul tipo puntato.

Puntatore NULL

- Un puntatore non inizializzato può dare qualche problema perché, di fatto, punta ad una locazione casuale in memoria.
- È quindi necessario avere un valore “neutro” da poter assegnare ad un puntatore per segnalare che non indirizza alcun dato valido in memoria.
- Per questo scopo si usa la costante simbolica **NULL** (definita in vari header tra cui **iostream**).

Puntatore NULL

- Un puntatore può essere inizializzato a

NULL:

```
int *p = NULL;
```

- È possibile operare un confronto con

NULL:

```
if (p != NULL)...
```

Puntatori e array

- Concetto di array molto vicino a quello di puntatore.
- L'identificatore di un array è equivalente all'indirizzo del primo elemento dell'array. Consideriamo il codice:

```
int vet[4]={5,7,9,2};  
int *p;
```

```
p=vet;  
cout << vet[0] << endl;  
cout << *p << endl;
```

- Quale la differenza tra il puntatore e la variabile array ? Entrambi contengono un indirizzo, ma:
 - **Il puntatore potrà modificare il suo valore e puntare ad un altro dato in memoria**
 - **La variabile array punterà sempre al primo dei 4 elementi int con cui è stata definita**
- Da questo punto di vista, una variabile array può essere considerata un **puntatore costante**.

Aritmetica dei puntatori

- È possibile compiere operazioni aritmetiche tra puntatori e interi.
- In particolare, sono ammesse:
 - Addizione di un puntatore ed un intero
 - Sottrazione di un intero da un puntatore
 - Differenza tra due puntatori

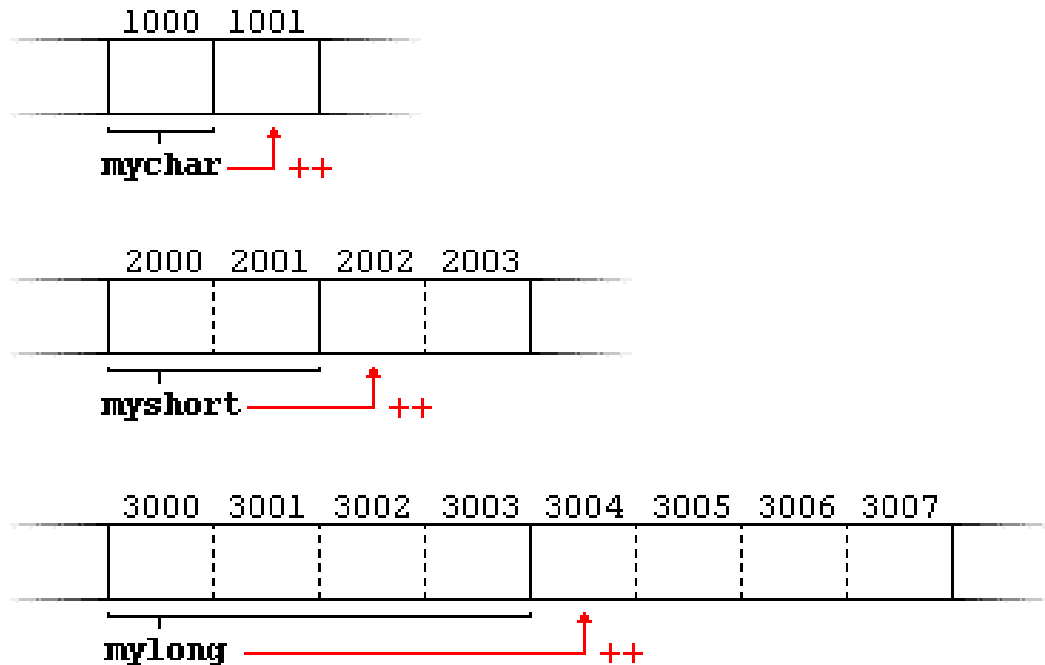
Aritmetica dei puntatori

- Che cosa significa sommare un puntatore ed un intero ? Qual è il risultato ?
- Il risultato è comunque un puntatore. A cosa ?
- Se un puntatore a un tipo T viene incrementato di 1, il suo valore viene di fatto incrementato di una quantità pari alla dimensione in byte del tipo T.

Aritmetica dei puntatori

```
char *mychar;  
short *myshort;  
long *mylong;
```

L'incremento individua
l'indirizzo della
variabile di tipo T
immediatamente
seguinte in memoria



© www.cplusplus.com

Aritmetica dei puntatori

- Lo stesso vale per il decremento.
- Che cosa succede se invece si somma (o si sottrae) un valore $k > 1$?
- $p+k$: il risultato è uguale al valore di p incrementato di $k*d$ dove d è la dimensione in byte del tipo puntato.
- Lo stesso vale per $p-k$

Puntatori e array /2

- Con l'aritmetica dei puntatori, il contatto tra puntatori e array diventa ancora più stretto.

```
int vet[4]={5,7,9,2};  
int *p;
```

```
p=vet;
```

- Infatti $p+1$ punterà a $v[1]$, $p+2$ a $v[2]$, ..., $p+i$ a $v[i]$
- Di fatto, $*(p+i)$ è un alias di $v[i]$.

Puntatori e array /2

- Che cosa realizza il codice seguente?

```
int vet[4]={5,7,9,2};  
int *p;  
  
p=vet;  
for(int i=0;i<4;i++)  
    cout << *(p+i) << endl;
```

Puntatori e array /2

- Tuttavia anche l'identificatore dell'array è un puntatore (costante) e quindi anche su `vet` si può adoprare l'aritmetica dei puntatori:

```
int vet[4]={5,7,9,2};  
  
for(int i=0;i<4;i++)  
    cout << *(vet+i) << endl;
```

- Qual è la differenza ?

Puntatori e array /2

- La grossa differenza di cui tener conto è che p è una variabile.

```
int vet[4]={5,7,9,2};  
int *p;
```

```
p=vet;  
for(int i=0;i<4;i++){  
    cout << *p << endl;  
    p = p+1;  
}
```


Puntatori e array /2

- Ai puntatori può essere applicato l'operatore di autoincremento:

```
for(int i=0;i<4;i++){  
    cout << *p << endl;  
    p++;  
}
```

- Si possono combinare autoincremento e dereferenziazione (non banale).

Aritmetica dei puntatori /2

- L'operatore di autoincremento (++) ha precedenza maggiore rispetto all'operatore di indirazione (*), ma assume un comportamento speciale quando è usato come suffisso: l'espressione è valutata con il valore che `p` ha prima di essere incrementato.
- Perciò l'espressione `*p++` che sarebbe equivalente a `*(p++)` di fatto è valutata come la sequenza:
`*p`
`p++`
- Si poteva fare diversamente ? Perché ? Che cosa succede con l'espressione `(*p)++` ?

Putting all together...

```
int vet[4]={5,7,9,2};  
int *p;  
  
p=vet;  
for(int i=0;i<4;i++)  
    cout << *p++ << endl;
```

Problema

- Scrivere una funzione di nome `transfer` che riceva in ingresso una stringa e trasferisca in altre due stringhe rispettivamente le vocali e le consonanti della stringa in ingresso.
- Implementare due versioni con
 - accesso agli elementi tramite indice
 - accesso agli elementi tramite puntatore

Puntatori a costanti

- È possibile definire puntatori a costanti, in modo che la variabile puntata non possa essere modificata tramite indirizione del puntatore.
- La dichiarazione prevede di anticipare la parola chiave `const` alla consueta dichiarazione. Es.:

```
const int *px;  
int x=3,y;
```

```
px=&x;
```

```
y=*px;
```

```
*px=5;
```

possibile

illegale

Puntatori come argomenti di funzione

- È possibile usare un puntatore come argomento di una funzione cui passare un riferimento o il valore di un puntatore come parametro effettivo.
- L'effetto netto è uguale a quanto visto per il passaggio per riferimento, solo che, con i puntatori, bisogna gestire esplicitamente l'indirizzazione.

Puntatori come argomenti di funzione

```
void scambia1(int &a, int &b)
{
    int appo;

    appo=a;
    a=b;
    b=appo;

    return;
}
```

```
void scambia2(int *a, int *b)
{
    int appo;

    appo=*a;
    *a=*b;
    *b=appo;

    return;
}
```

Puntatori come argomenti di funzione

- L'uso dei puntatori come argomenti può essere vantaggioso quando si lavora con gli array.
- In effetti, l'array parametro formale viene gestito come un puntatore.
- Es.: riscrivere la funzione `transfer` usando puntatori come argomenti.

Puntatori a strutture

- Come le altre variabili, anche per le variabili struttura è possibile definire puntatori.
- L'unica particolarità è l'accesso ai campi di una struttura riferita tramite puntatore:

```
Studente s1 = {"Paolino",  
              "Paperino",  
              1234,  
              "LINF TLC"};
```

```
Studente *pstud;  
pstud = &s1;
```

Parentesi
necessarie



```
cout << "Cognome: " << (*pstud).cognome <<  
endl;
```

Puntatori a strutture

- È necessario racchiudere tra parentesi () il puntatore dereferenziato perché l'operatore . (punto) ha la precedenza rispetto all'operatore di indirizione.
- È possibile però utilizzare l'operatore apposito '->' per cui sono equivalenti le espressioni (*pstud).cognome e pstud->cognome :

```
cout << "Cognome: " << pstud->cognome << endl;
```

Puntatori a funzioni

- Il C++ permette la definizione di puntatori a funzioni. In questo caso non si considera l'indirizzo di una variabile, ma l'indirizzo della prima istruzione di una funzione.
- L'impiego tipico di un puntatore a funzione è il passaggio di una funzione come argomento ad un'altra funzione.
- La definizione di un puntatore a funzione è simile al prototipo della funzione, tranne che il nome della funzione (in effetti, il nome del puntatore) è racchiuso tra parentesi () ed è preceduto da un asterisco:
*tipo_restituito (*nome_puntatore) (lista_argomenti);*

- Esempio:
`int (*pfun)(int x, int y);`

definisce il puntatore `pfun` ad una funzione che ha due argomenti `int` passati per valore e restituisce un `int`.

Puntatori a funzioni

- Per inizializzare un puntatore a funzione, è sufficiente assegnare il nome della funzione al puntatore.

```
int somma(int x, int y){  
    return x+y;  
}
```

```
int main() {  
    int (*pf)(int,int); ← Definizione  
                           del puntatore  
    pf = somma; ← Assegnazione  
                  del puntatore  
}
```

Puntatori a funzioni

- Per dereferenziare il puntatore (e invocare la funzione puntata) si scrive tra parentesi () il nome del puntatore preceduto da un asterisco e quindi la lista dei parametri effettivi tra parentesi ().

```
int somma(int x, int y){  
    return x+y;  
}
```

```
int main() {  
    int a=1,b=2,c;  
    int (*pf)(int,int);
```

```
    pf = somma;  
    c = (*pf)(a,b);  
}
```

← Indirezione
del puntatore

```
#include <iostream>
using namespace std;

int somma(int x, int y);
int diff(int x, int y);
int operazione(int (*pfun)(int,int),int x, int y);

int main(int argc, char** argv) {
    int a,b,c;

    cout << "Primo valore: "; cin >> a;
    cout << "Secondo valore: "; cin >> b;

    if(a > b)
        c = operazione(diff,a,b);
    else
        c = operazione(somma,a,b);

    cout << "Risultato: " << c << endl;
    return (EXIT_SUCCESS);
}
```

[segue... →](#)

```
int somma(int x, int y){  
    return x+y;  
}
```

```
int diff(int x, int y){  
    return x-y;  
}
```

```
int operazione(int (*pfun)(int,int),int x, int y){  
    int z;  
  
    z = (*pfun)(x,y);  
  
    return z;  
}
```