



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - Puntatori (parte prima)

Aggiornamento del 17/10/2019

I puntatori

- Abbiamo visto come, nell'architettura ADE8, sia stato necessario introdurre il concetto di accesso indiretto per consentire di lavorare con più dati di cui era noto solo l'indirizzo iniziale.
- Per fare questo è stato necessario utilizzare il valore contenuto in una cella come indirizzo.
- In C non sarebbe sufficiente un semplice tipo «indirizzo». Quello che bisogna specificare è anche il tipo di dato di cui si gestisce l'indirizzo.
- In C esiste quindi **il puntatore ad un tipo di dato**. Ovvero avremo puntatori a char, puntatori ad unsigned char, puntatori a short, puntatori ad unsigned short, puntatori a int, ...
- Un puntatore quindi è il tipo di dato che rappresenta l'indirizzo di una variabile di un tipo ben specificato.
- La dimensione in memoria di un puntatore non dipende dal tipo di dato a cui punta, dato che tutti gli indirizzi sono uguali e dipendono dall'architettura.
- A meno di non cambiare il default, su Visual Studio 2017 i puntatori sono a 32 bit.

Definizione di un puntatore

- Un puntatore viene definito come una qualsiasi variabile, specificando il tipo del dato a cui si punta e premettendo al nome della variabile il modificatore *

```
<tipo> *<nome variabile>;
```

- Attenzione quindi al fatto che **l'asterisco è un modificatore della variabile** e non del tipo. Si può scrivere quindi:

```
int *a;
```

```
int* b;
```

```
int*c;
```

- e tutte e tre sono puntatori a int. Nel caso:

```
char* p, q; ← Attenzione
```

- p è un puntatore a un intero a 8 bit con segno (e occuperà 4 byte), mentre q è un intero a 8 bit con segno (e occuperà 1 byte). **Mettete l'asterisco attaccato alla variabile che modifica!**

Utilizzo dei puntatori: operatore &

- Supponiamo di avere una variabile `x` di tipo `char` e voler puntare a quella variabile con il puntatore `p`. Come faccio ad assegnare a `p` l'indirizzo di `x`? Mi serve un operatore che ritorni l'indirizzo di `x`:

`&<nome variabile>`

- L'operatore unario `&` restituisce l'indirizzo della variabile a cui è applicato. Ad esempio possiamo fare:

```
int main(void)
{
    char x = 3;
    char *p;

    p = &x;
    return 0;
}
```

- In questo modo `p` è una variabile di tipo puntatore che contiene l'indirizzo di `x`. Si dice che «`p` punta a `x`».

Utilizzo dei puntatori: operatore *

- Ok, ma a che cosa serve avere un puntatore? Come posso cioè utilizzarlo?
- Supponiamo ad esempio di voler leggere il valore della memoria all'indirizzo puntato da p. Come fare? Bisogna utilizzare un operatore che, dato il puntatore, restituisca *la variabile puntata*:

`*<variabile di tipo puntatore>`

- Ma come? Non abbiamo già visto che l'asterisco serve per *definire* una variabile di tipo puntatore? Purtroppo sì, cioè in C l'asterisco, come visto per le parentesi tonde, viene utilizzato in diversi modi e a seconda del caso assume significati diversi:
 - in una espressione come `a*b`, è l'operatore binario di moltiplicazione;
 - in una espressione come `a = *b`, è l'operatore unario di *dereferenziazione*. Ovvero, l'espressione `*b` è il valore della variabile all'indirizzo b.
 - In una definizione come `int *a;`, è il modificatore che definisce a come puntatore a int.

Utilizzo dell'operatore di dereferenziazione

- Quando si usa l'operatore *, si ottiene la variabile puntata dal puntatore e quindi si può operare su quella variabile tramite il puntatore. Ad esempio:

```
int main(void)
{
    char x = -27;
    char *p = &x;
    int b;

    *p = -80;
    b = x;

    return 0;
}
```

- Quanto vale b quando l'esecuzione raggiunge il comando return?
- Il valore di x.
- Che è diventato -80, dopo l'assegnamento a *p. Infatti p punta a x, quindi *p è la zona di memoria occupata da x. Una modifica a *p modifica esattamente la memoria di x, quindi modifica x.
- Quello di puntatore è un concetto molto importante e non di immediata comprensione, quindi è opportuno sforzarsi di capirlo bene e il prima possibile!

Applicazione dei puntatori

- Riprendiamo l'esempio **sbagliato** che abbiamo già visto in precedenza, dove tentavamo di scrivere una funzione in grado di raddoppiare un valore numerico:

```
int y = 7;
```

```
void raddoppia (int x) {  
    x = x * 2;  
}
```

← Programma errato di esempio

```
int main(void) {  
    raddoppia(y);  
    return 0;  
}
```

- Abbiamo già detto che al momento della chiamata alla funzione raddoppia, viene creato il parametro x della funzione inizializzato al valore di y. Quindi modificare il valore di x non ha alcun effetto sulla cella di memoria che chiamiamo y.
- Possiamo «legare» le due variabili trasformando x in un puntatore.

Applicazione dei puntatori

- La versione funzionante della funzione è la seguente:

```
int y = 7;
```

```
void raddoppia (int *x) {  
    *x = *x * 2;  
}
```

```
int main(void) {  
    raddoppia(&y);  
    return 0;  
}
```

- Non è proprio bella da leggere o da usare, ma funziona:
 - Invece che passare a raddoppia il valore di y, passo l'indirizzo;
 - Nella funzione dereferenzio (neologismo...) il parametro con l'operatore * e quindi leggo e scrivo le celle di memoria in cui è contenuto y.
 - Al termine della funzione, la variabile x viene distrutta, ma tanto le modifiche sono state applicate direttamente a y.
- In generale, potendo, la soluzione che non utilizza puntatori è preferibile!

Applicazione dei puntatori

- Un caso in cui i puntatori si rendono veramente utili è quello in cui ci sia bisogno di ritornare più valori da una funzione, oppure quando più variabili sono coinvolte nei calcoli e debbano essere modificate.
- Il caso classico è lo scambio di due variabili.
- Riprendiamo la funzione MCD con scambio:

```
unsigned int MCD (unsigned int m, unsigned int n) {  
    if (m == 0 || n == 0)  
        return 0;  
    while (m != n) {  
        if (m < n) {  
            unsigned int t = m;  
            m = n;  
            n = t;  
        }  
        m -= n;  
    }  
    return m;  
}
```

- Il corpo della funzione è riempito principalmente dallo scambio di m e n.
- Potrebbe essere più chiaro utilizzare una funzione per questo scopo.
- Vediamone una versione sbagliata.

Applicazione dei puntatori

- Ecco cosa potrebbe pensare di scrivere un programmatore distratto:

```
void swap(unsigned int a, unsigned int b) {  
    unsigned int tmp = a;  
    a = b;  
    b = tmp;  
}
```

← Programma errato di esempio

```
unsigned int MCD(unsigned int m, unsigned int n) {  
    if (m == 0 || n == 0)  
        return 0;  
    while (m != n) {  
        if (m < n)  
            swap(m, n);  
        m -= n;  
    }  
    return m;  
}
```

- Ovviamente questa funzione non fa nulla!!! I valori di a e b vengono scambiati, ma quelli di m e n rimarranno esattamente identici a prima.
- Quello che si deve fare è utilizzare i puntatori per modificare le variabili m e n.

Applicazione dei puntatori

- La versione corretta è la seguente:

```
void swap(unsigned int *a, unsigned int *b) {
    unsigned int tmp = *a;
    *a = *b;
    *b = tmp;
}

unsigned int MCD(unsigned int m, unsigned int n) {
    if (m == 0 || n == 0)
        return 0;
    while (m != n) {
        if (m < n)
            swap(&m, &n);
        m -= n;
    }
    return m;
}
```

- Qui, passiamo alla funzione gli indirizzi delle variabili da scambiare e tramite questi indirizzi, modifichiamo il contenuto della memoria di quelle variabili.

Visualizzazione grafica dell'operazione di swap

- Proviamo a dare una interpretazione grafica dei puntatori visualizzando con dei rettangoli le celle di memoria e con delle frecce il fatto che queste puntino ad altre celle.
- Inizialmente questa è la situazione:

MCD

m	3
n	6

Visualizzazione grafica dell'operazione di swap


- Viene invocata la funzione swap e questo crea le nuove variabili a e b:

MCD

m	3
n	6

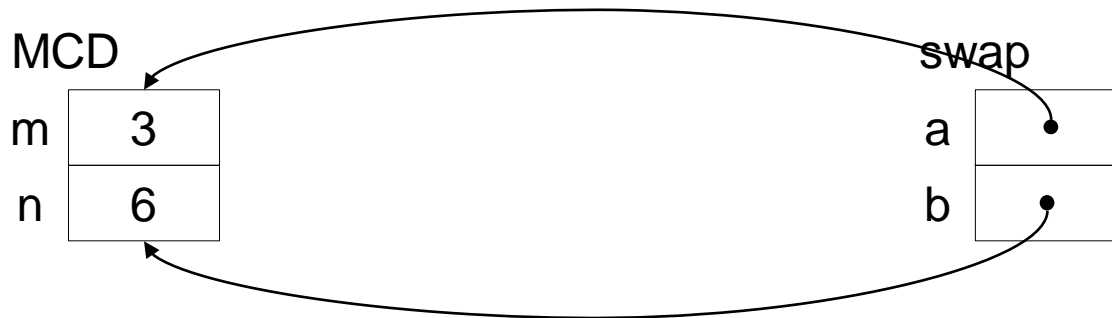
swap

a	?
b	?

```
void swap(unsigned int *a, unsigned int *b) {   
    unsigned int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Visualizzazione grafica dell'operazione di swap

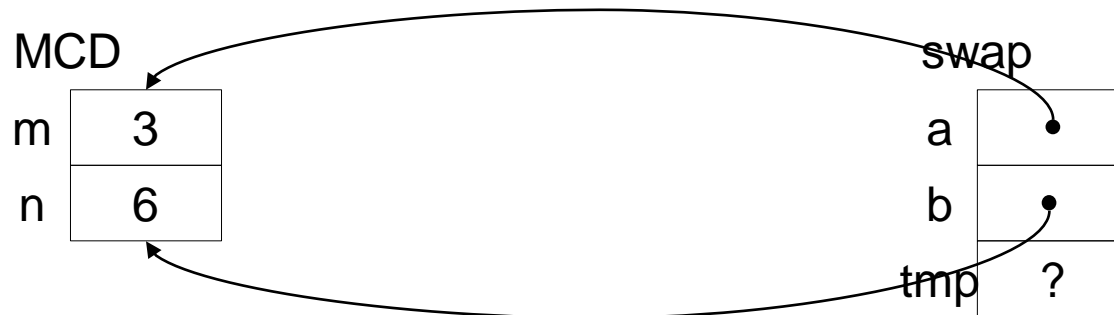
- Viene invocata la funzione swap e questo crea le nuove variabili a e b:
- Poi avviene l'inizializzazione di a e b con &m e &n:



```
void swap(unsigned int *a, unsigned int *b) { ←  
    unsigned int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Visualizzazione grafica dell'operazione di swap

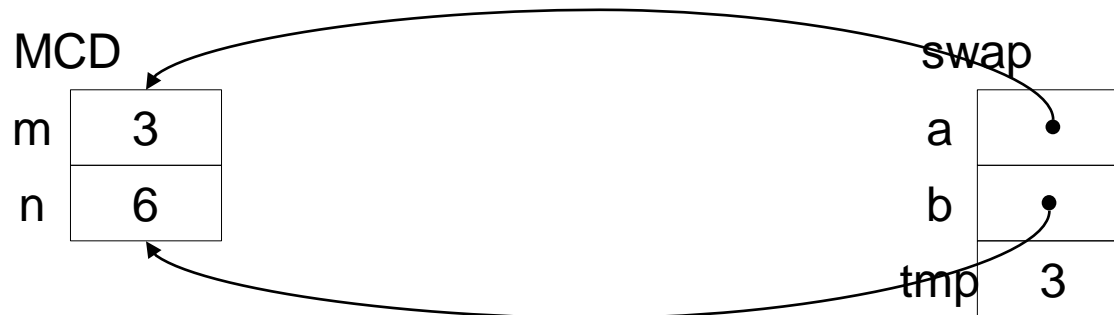
- Viene invocata la funzione swap e questo crea le nuove variabili a e b.
- Poi avviene l'inizializzazione di a e b con &m e &n.
- Si entra nel corpo della funzione e viene creata la variabile tmp:



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int tmp = *a; ←  
    *a = *b;  
    *b = tmp;  
}
```

Visualizzazione grafica dell'operazione di swap

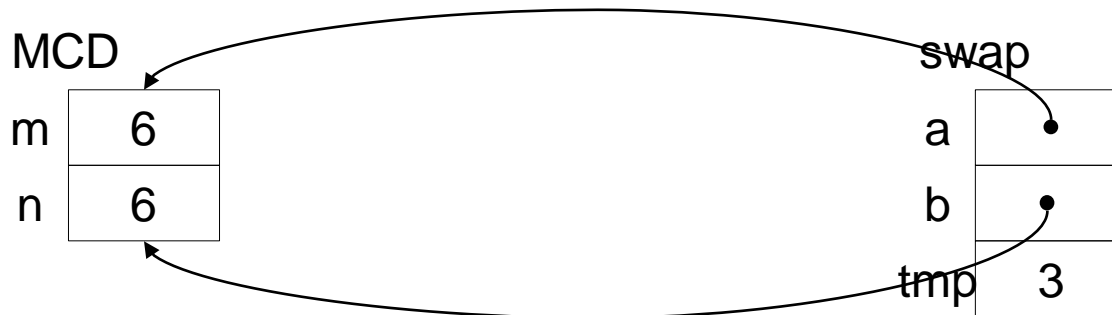
- Viene invocata la funzione swap e questo crea le nuove variabili a e b.
- Poi avviene l'inizializzazione di a e b con &m e &n.
- Si entra nel corpo della funzione e viene creata la variabile tmp.
- tmp viene inizializzata con *a, ovvero il valore puntato da a (3):



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int tmp = *a; ←  
    *a = *b;  
    *b = tmp;  
}
```


Visualizzazione grafica dell'operazione di swap

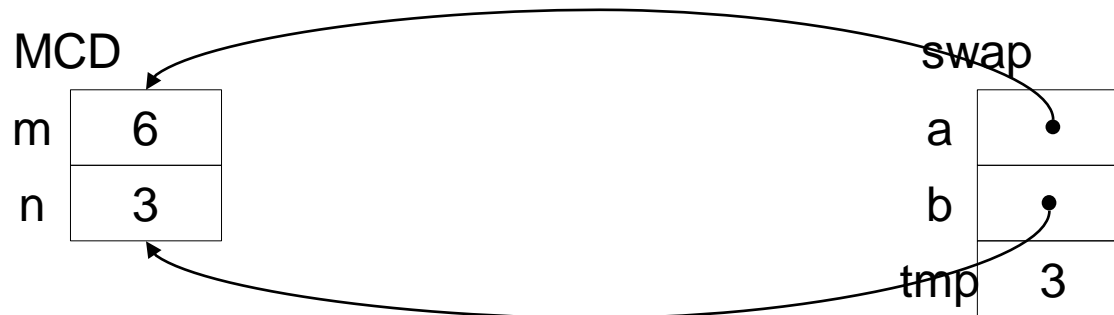
- Alla cella puntata da a si assegna il valore di quella puntata da b (6):



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int tmp = *a;  
    *a = *b; ←  
    *b = tmp;  
}
```

Visualizzazione grafica dell'operazione di swap

- Alla cella puntata da a si assegna il valore di quella puntata da b (6).
- Alla cella puntata da b si assegna il valore contenuto in tmp (3):



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int tmp = *a;  
    *a = *b;  
    *b = tmp; ←  
}
```


Visualizzazione grafica dell'operazione di swap

- Alla cella puntata da a si assegna il valore di quella puntata da b (6).
- Alla cella puntata da b si assegna il valore contenuto in tmp (3).
- Al termine della funzione, le variabili e i parametri vengono eliminati e si ritorna alla funzione chiamante:

MCD

m	6
n	3

swap

```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int tmp = *a;  
    *a = *b;  
    *b = tmp;  
} 
```

Un possibile errore

- Lavorando coi puntatori, è facile confondersi e cambiare il valore del puntatore (cioè farlo puntare da un'altra parte) invece che il valore della cella puntata:

```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;  
    a = b;  
    b = tmp;  
}
```

← Programma errato di esempio

```
unsigned int MCD(unsigned int m, unsigned int n) {  
    if (m == 0 || n == 0)  
        return 0;  
    while (m != n) {  
        if (m < n)  
            swap(&m, &n);  
        m -= n;  
    }  
    return m;  
}
```

- Questa funzione sembra fare quanto richiesto, ma all'uscita, di nuovo, nulla è cambiato.

Visualizzazione grafica della **swap sbagliata**

- Inizialmente questa è la situazione:

MCD

m	3
n	6

Visualizzazione grafica della **swap sbagliata**


- Viene invocata la funzione swap e questo crea le nuove variabili a e b:

MCD

m	3
n	6

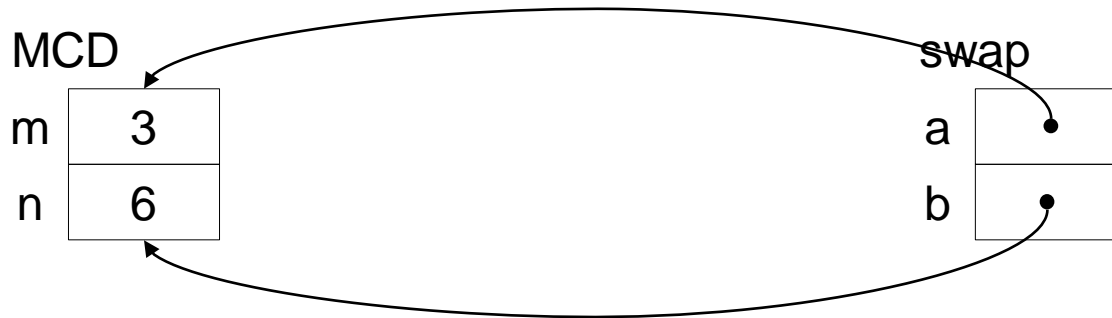
swap

a	?
b	?

```
void swap(unsigned int *a, unsigned int *b) {   
    unsigned int *tmp = a;  
    a = b;  
    b = tmp;  
}
```

Visualizzazione grafica della **swap sbagliata**

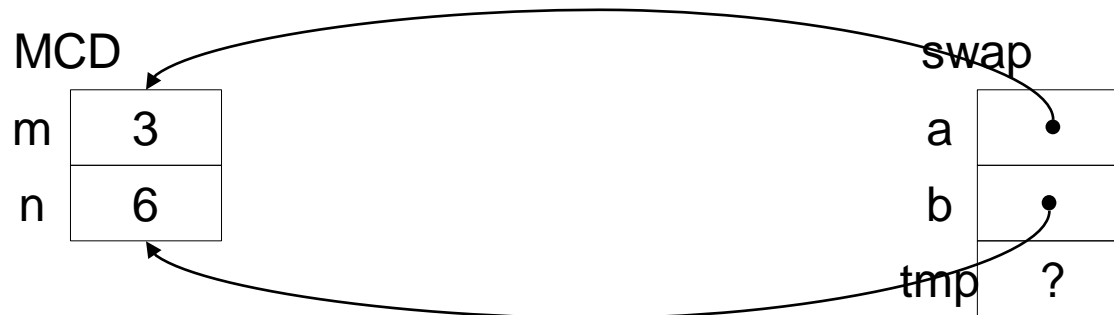
- Viene invocata la funzione swap e questo crea le nuove variabili a e b:
- Poi avviene l'inizializzazione di a e b con &m e &n:



```
void swap(unsigned int *a, unsigned int *b) { ←  
    unsigned int *tmp = a;  
    a = b;  
    b = tmp;  
}
```

Visualizzazione grafica della **swap sbagliata**

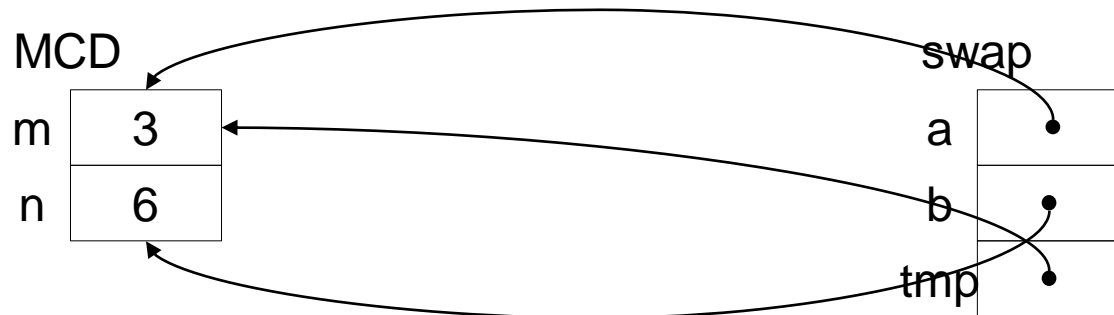
- Viene invocata la funzione swap e questo crea le nuove variabili a e b.
- Poi avviene l'inizializzazione di a e b con &m e &n.
- Si entra nel corpo della funzione e viene creata la variabile tmp:



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;   
    a = b;  
    b = tmp;  
}
```


Visualizzazione grafica della **swap sbagliata**

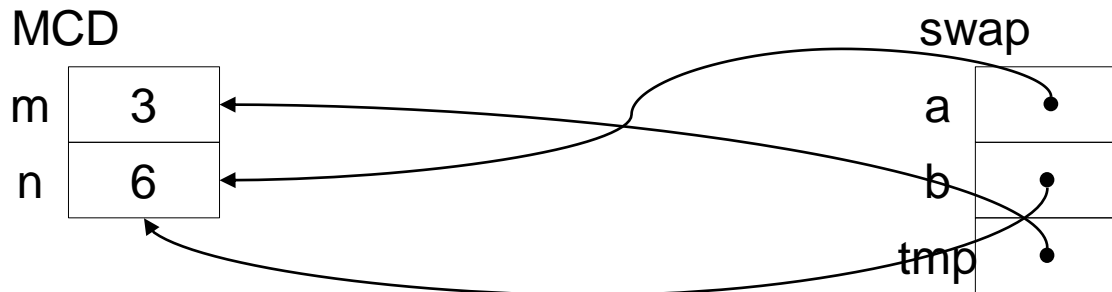
- Viene invocata la funzione swap e questo crea le nuove variabili a e b.
- Poi avviene l'inizializzazione di a e b con &m e &n.
- Si entra nel corpo della funzione e viene creata la variabile tmp.
- tmp viene inizializzata con a, ovvero l'indirizzo puntato da a (&m):



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;   
    a = b;  
    b = tmp;  
}
```

Visualizzazione grafica della **swap sbagliata**

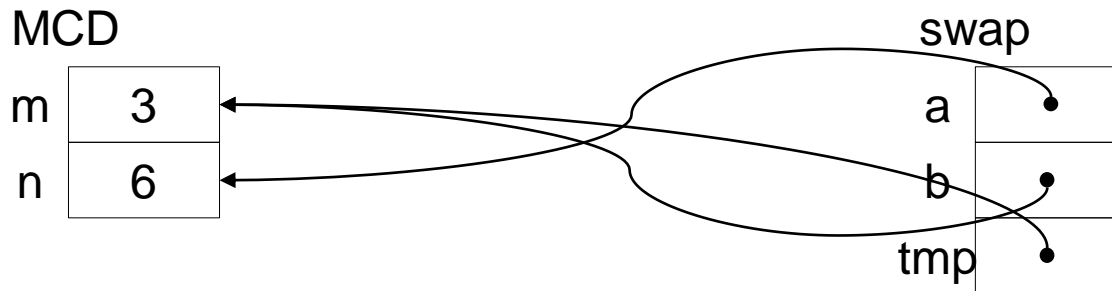
- Ad a si assegna il valore di b, ovvero si fa puntare a dove punta anche b:



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;  
    a = b; ←  
    b = tmp;  
}
```

Visualizzazione grafica della **swap sbagliata**

- Ad a si assegna il valore di b, ovvero si fa puntare a dove punta anche b.
- A b si assegna il valore contenuto in tmp, ovvero b punta dove punta anche tmp:



```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;  
    a = b;  
    b = tmp; ←  
}
```

Visualizzazione grafica della **swap sbagliata**

- Ad a si assegna il valore di b, ovvero si fa puntare a dove punta anche b.
- A b si assegna il valore contenuto in tmp, ovvero b punta dove punta anche tmp.
- Al termine della funzione, le variabili e i parametri vengono eliminati e si ritorna alla funzione chiamante:

MCD

m	3
n	6

swap

- **Purtroppo m e n non sono state modificate! La funzione in pratica non fa nulla!**

```
void swap(unsigned int *a, unsigned int *b) {  
    unsigned int *tmp = a;  
    a = b;  
    b = tmp;  
} ←
```

Attenzioni da avere

- Ogni volta che si utilizzano i puntatori, è importante fare attenzione al significato che questi hanno.
- Un puntatore è un numero (come tutto, del resto), che indica dove si trova una certa variabile in memoria.
- Cambiare il numero contenuto nel puntatore (assegnare al puntatore qualcosa) significa farlo puntare da un'altra parte.
- Per leggere o modificare la cella puntata bisogna utilizzare l'operatore di dereferenziazione.
- Attenzione anche ai tipi:
 - se abbiamo la variabile x di tipo T , l'espressione $\&x$ è di tipo T^* (puntatore a T);
 - se abbiamo la variabile p di tipo T^* , l'espressione $*p$ è di tipo T (la variabile puntata da p).
 - se abbiamo la variabile p di tipo T^* , l'espressione $\&p$ è di tipo T^{**} (puntatore a puntatore a T ... argh!).
 - se abbiamo la variabile x di tipo T , l'espressione $*x$ non ha senso (errore)!

Altre accortezze

- Consideriamo di avere a disposizione la funzione seguente:

```
int divmod(int num, int den, int *q, int *r) {  
    if (den == 0)  
        return 0;  
  
    *q = num / den;  
    *r = num % den;  
    return 1;  
}
```

- Questa funzione accetta due int come parametri e calcola il quoziente e il resto della divisione di num per den, se den è diverso da 0. Se den è 0 restituisce 0 e non fa alcun calcolo.
- Scriviamo un main che utilizzi questa funzione. Lo studente distratto dice: «devo passargli due int e due puntatori a int, quindi li dichiaro e glieli passo».
- E quindi produce il programma seguente...

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}  
  
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c, d);  
    return 0;  
}
```

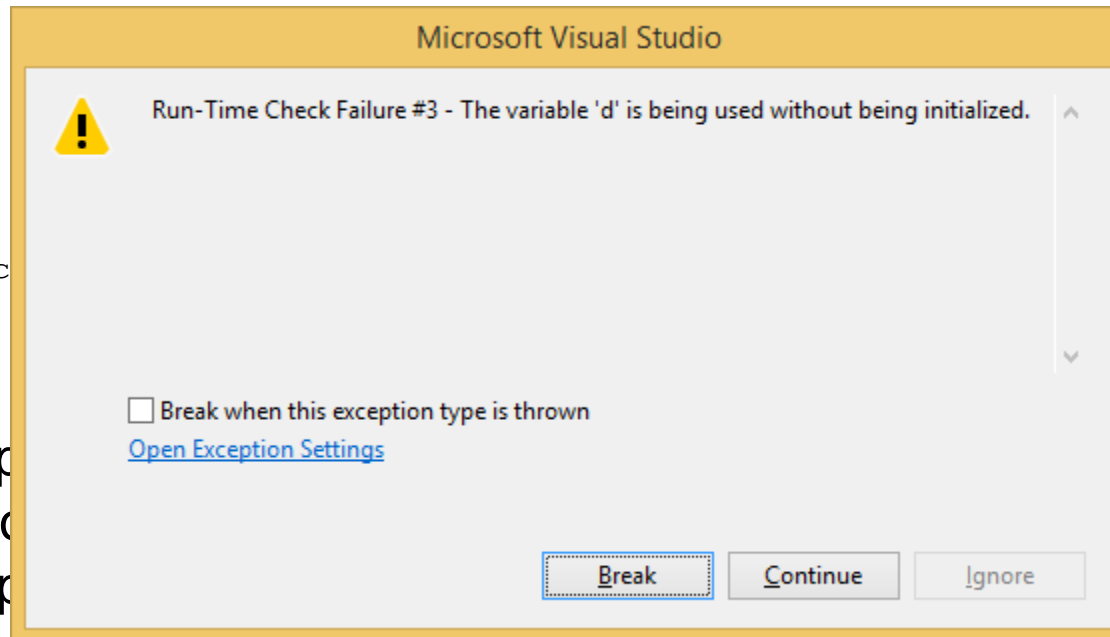
← Programma errato di esempio

- Già in compilazione ci sono alcuni warning (avvertimenti che solitamente indicano problemi). Ma lo studente testardamente insiste e prova ad eseguire il programma.
- Arrivato a divmod, cosa succede?

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}  
  
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c,  
    return 0;  
}
```

- Già in compilazione, gli errori di tipo indicano problemi che possono verificarsi durante l'esecuzione del programma.
- Arrivato a `divmod`, cosa succede?



errato di esempio

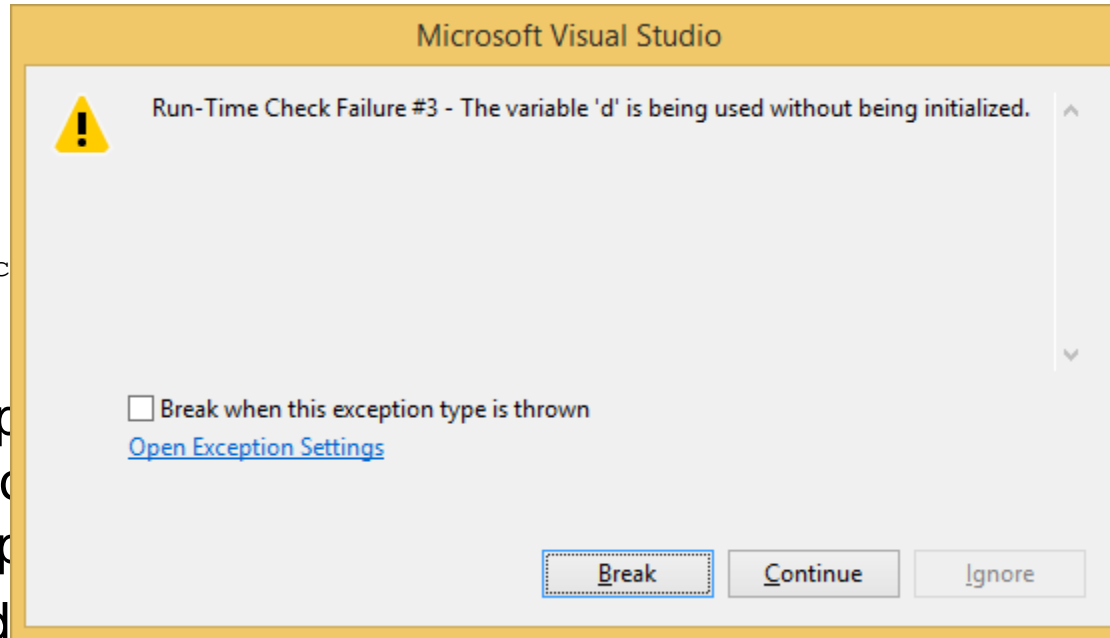
che solitamente
prova ad

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}
```

```
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c,  
    return 0;  
}
```

- Già in compilazione, i messaggi di errore indicano problemi. In questo caso, il compilatore segnala che la variabile `d` è usata senza essere stata inizializzata.
- Arrivato a debug, il Visual Studio ci avverte che la variabile `d` è usata senza essere stata inizializzata.



...arrivato di esempio

...e solitamente
prova ad

Va beh, tanto non volevo inicializzarla. Noioso di un Visual Studio...

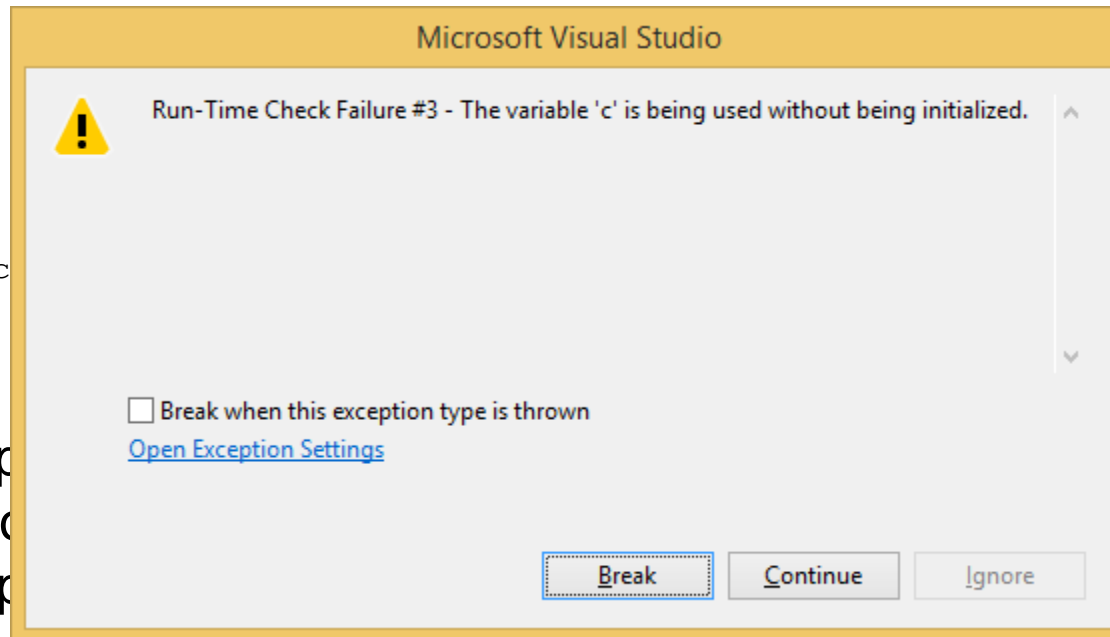
Premiamo Continue e speriamo bene...

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}
```

```
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c, d);  
    return 0;  
}
```

- Già in compilazione, i messaggi di errore indicano problemi. In questo caso, non si può eseguire il programma.
- Arrivato a `divmod`, cosa succede?



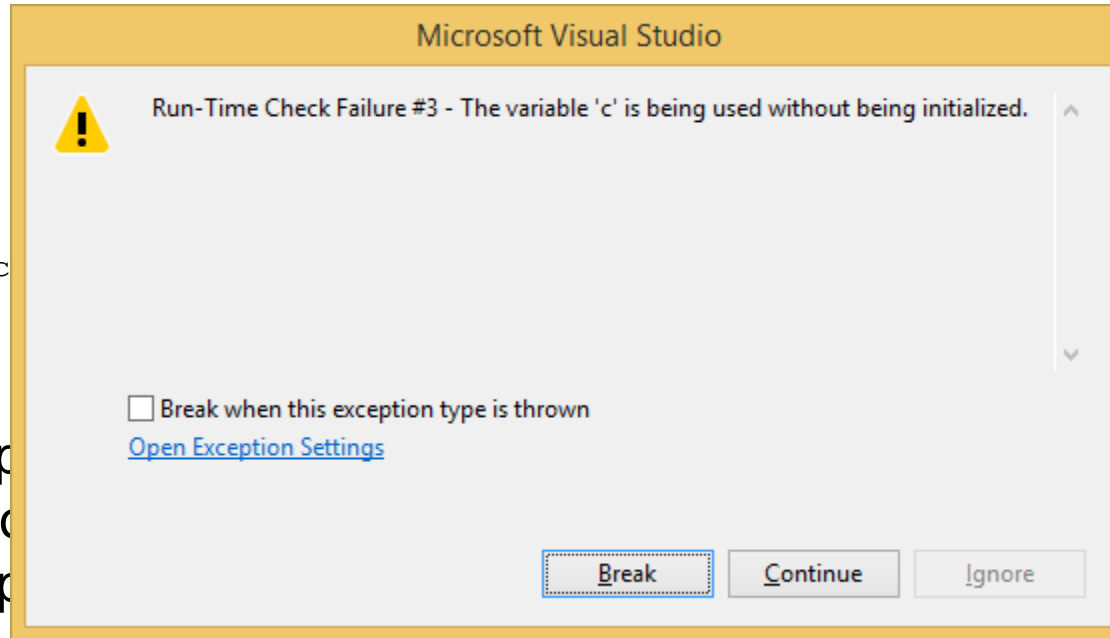
Arrivato di esempio

che solitamente
prova ad

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}  
  
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c, d);  
    return 0;  
}
```

- Già in compilazione, gli errori indicano problemi che possono impedire di eseguire il programma.
- Arrivato a divmod, cosa succede?



errato di esempio

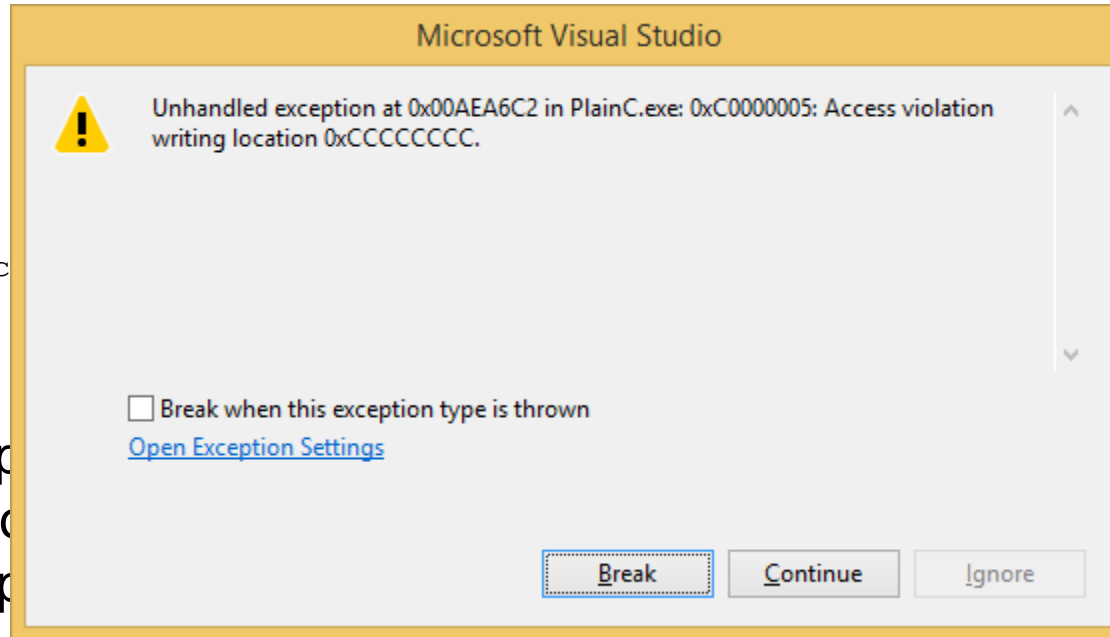
che solitamente
prova ad

Ancora?! Noioso di un Visual Studio...
Premiamo Continue e speriamo bene...

Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}  
  
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c, d);  
    return 0;  
}
```

- Già in compilazione, gli errori di tipo indicano problemi che possono impedire di eseguire il programma.
- Arrivato a `divmod`, cosa succede?



errato di esempio

che solitamente
prova ad

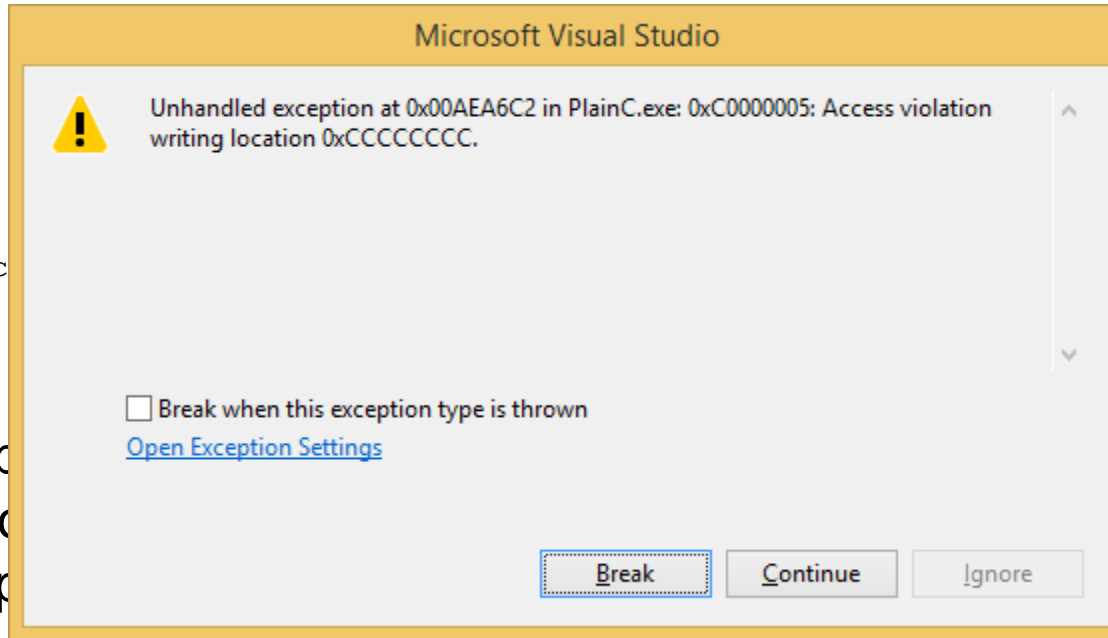
Altre accortezze

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}
```

```
int main(void) {  
    int a = 135;  
    int b = 7;  
    int *c;  
    int *d;  
  
    divmod(a, b, c, d);  
    return 0;  
}
```

- Già in compilazione, gli errori indicano problemi che possono impedire di eseguire il programma.

- Arrivato a `divmod`, cosa succede?



errato di esempio

che solitamente
prova ad

Argh, qui Continue non funziona più...

Qual è il problema?

- Il problema è che nel nostro main abbiamo dichiarato due puntatori non inizializzati. A che cosa puntano? Boh... (questa è la risposta corretta).
- Bisogna farli puntare a due variabili che possano contenere degli int. Il main corretto può quindi essere:

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}
```

```
int main(void) {  
    int a = 135;  
    int b = 7;  
    int c, d;  
    int *pc = &c;  
    int *pd = &d;  
  
    divmod(a, b, pc, pd);  
    return 0;  
}
```

- Cioè definiamo due variabili che possono contenere due int e due puntatori che puntano a quelle variabili. Poi le passiamo alla funzione.
- Ma servono quelle variabili? No, sono solo usate per avere dei puntatori. Ma le espressioni &c e &d sono già puntatori a int, quindi...

Come si risolve

- Usiamo direttamente gli indirizzi di c e d:

```
int divmod(int num, int den, int *q, int *r) {  
    ...  
}  
  
int main(void) {  
    int a = 135;  
    int b = 7;  
    int c, d;  
  
    divmod(a, b, &c, &d);  
    return 0;  
}
```

- È importante ricordare che **se una funzione accetta un parametro di tipo puntatore a qualcosa, non è sempre necessario definire una variabile di quel tipo!** Quello che bisogna fare è passare alla funzione una espressione di tipo puntatore a qualcosa.
- I puntatori devono puntare a qualcosa prima di essere utilizzati, altrimenti puntano in un posto a caso in memoria e questo produce risultati a dir poco sconcertanti.