



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - break, continue, switch, enum,
union

Aggiornamento del 12/11/2018

Ricerca della maiuscola

- Supponiamo di dover scrivere una funzione che prenda come parametro una stringa `C` e che sia in grado di trovare la prima lettera maiuscola dopo un punto.
- La funzione prende come unico parametro un puntatore al primo char della stringa e ritorna un int contente la posizione della prima lettera maiuscola dopo un punto. Se questo caso non si verifica mai allora ritorna -1 per indicare il fallimento.
- Per scrivere questa funzione possiamo sfruttare la libreria `ctype.h` e la funzione `isupper()` per controllare se un carattere è maiuscolo oppure no.
- La dichiarazione della funzione potrebbe essere la seguente:

```
extern int cerca_maiuscola(const char *s);
```

- L'unico parametro in questo caso è il puntatore al primo elemento della stringa (non dovendola modificare il puntatore è anche `const`).

Ricerca della maiuscola

- Il codice per realizzare questa funzione potrebbe essere il seguente:

```
#include <stdlib.h>
#include <ctype.h>
```

```
int cerca_maiuscola(const char *s) {
    size_t pos = 0;
    int maiusc = -1;

    while (s[pos] != 0) {
        if (s[pos] == '.' && isupper(s[pos + 1]))
            maiusc = pos + 1;

        pos++;
    }

    return maiusc;
}
```

```
int main(void) {
    char str[] = "ciao.Come stai?";
    int pos = cerca_maiuscola(str);

    return 0;
}
```

Scorro la stringa finché non incontro il terminatore.

Se il carattere corrente è un punto e il successivo è maiuscolo allora assegno a maiusc la posizione corrente più uno.

Incremento il contatore della posizione.

Ritorno la posizione in cui ho trovato la maiuscola.

Ricerca della maiuscola

- Questa funzione, quando prende come input la stringa "ciao.Come stai?" funziona correttamente ma potremmo trovare già due importanti problemi in questo codice.
- Il primo e più importante riguarda l'efficacia della funzione. Esso è dato dal fatto che non abbiamo considerato cosa succede se nella stringa sono presenti più casi di punti seguiti da lettere maiuscole.
- In quel caso la funzione non ritornerebbe la posizione della prima maiuscola che segue un punto ma dell'ultima!
- Il secondo problema riguarda l'efficienza della funzione. Essa infatti continua a scorrere la stringa anche dopo aver individuato la prima maiuscola.
- Se la stringa fosse molto lunga sarebbe uno spreco notevole di risorse.

Ricerca della maiuscola con goto

- La soluzione per evitare questi due problemi è *interrompere* il ciclo che cerca la maiuscola non appena ha trovato la prima.
- Il primo modo che potremmo usare potrebbe essere quello di usare un goto e un'etichetta per saltare alla fine del ciclo, analogamente a quello che facevamo nell'assembly di ADE8 , ottenendo il seguente codice:

```
int cerca_maiuscola(const char *s) {  
    size_t pos = 0;  
    int maiusc = -1;  
  
    while (s[pos] != 0) {  
        if (s[pos] == '.' && isupper(s[pos + 1])) {  
            maiusc = pos + 1;  
            goto trovata;  
        }  
  
        pos++;  
    }  
    trovata:  
    return maiusc;  
}
```

Uso un goto per saltare al comando return della funzione. Per farlo ho creato un'etichetta chiamata «trovata».

Break

- Ma abbiamo come abbiamo già detto il goto è da evitare!
- Per fortuna in C esiste un altro modo per ottenere lo stesso risultato, il comando **break**.
- La sintassi per utilizzarlo è data semplicemente dalla keyword break:

break ;

- Questo comando, quando incontrato all'interno di un ciclo (for, while e do-while), sposta l'esecuzione al primo comando dopo il corpo del ciclo.
- Il comando break può essere usato *solo* all'interno di un ciclo (e in un altro comando che vedremo più avanti).
- Il comportamento è analogo a quello di un goto applicato come nella slide precedente.
- Al contrario del goto però, il suo significato è più chiaro non lascia spazio a errori e disattenzioni che possono portare a comportamenti imprevisti (soprattutto per chi legge e/o lavora sul codice scritto da altri).

Break

- La funzione precedente, modificata per usare il break, diventerà quindi:

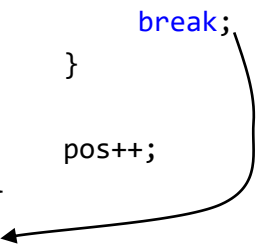
```
int cerca_maiuscola(const char *s) {  
    size_t pos = 0;  
    int maiusc = -1;  
  
    while (s[pos] != 0) {  
        if (s[pos] == '.' && isupper(s[pos + 1])) {  
            maiusc = pos + 1;  
            break;  
        }  
  
        pos++;  
    }  
  
    return maiusc;  
}
```

- Al posto del goto è stato inserito un break e l'etichetta che era stata messa appena dopo il ciclo while è sparita, in quanto non serve più.

Break

- Quando si verifica la condizione dell'if, dopo aver assegnato a `maiusc` la posizione corrente più uno l'esecuzione si sposta al primo comando dopo la fine del ciclo.

```
int cerca_maiuscola(const char *s) {  
    size_t pos = 0;  
    int maiusc = -1;  
  
    while (s[pos] != 0) {  
        if (s[pos] == '.' && isupper(s[pos + 1])) {  
            maiusc = pos + 1;  
            break;  
        }  
        pos++;  
    }  
    return maiusc;  
}
```



A curved arrow originates from the `break;` statement inside the `while` loop and points to the `return maiusc;` statement, illustrating that the loop is exited immediately and control jumps to the first statement following the loop.

- In questo caso il primo (e unico) comando successivo all'esecuzione del ciclo è il comando `return`.

Continue

- Durante l'esecuzione di un ciclo, un'altra situazione che si potrebbe incontrare, è quella di dover *evitare* di eseguire la restante parte del corpo del ciclo se si verifica una certa condizione.
- Supponiamo ad esempio di dover scrivere una funzione che, data una stringa contenente un numero decimale sia in grado di ritornare la somma delle sue cifre, sia della parte intera che della parte decimale.
- La dichiarazione della funzione potrebbe essere questa:

```
extern size_t somma_cifre_dec(const char *s);
```

- La funzione ha un solo parametro, un puntatore a char (anche in questo caso è `const`, dato che non dobbiamo modificare la stringa) e ritorna un `size_t` contenente la somma delle cifre.

Continue

- In un primo tentativo potrei scrivere quanto segue:

```
#include <stdlib.h>
```

```
size_t somma_cifre_dec(const char *s) {  
    size_t i, somma = 0;
```

```
    for (i = 0; s[i] != 0; ++i) {  
        somma += s[i] - '0';  
    }
```

```
    return somma;  
}
```

```
int main(void) {  
    char *str = "-143.678";  
    size_t somma = somma_cifre_dec(str);  
  
    return 0;  
}
```

Scorro con un for tutti i caratteri della stringa finché non incontro il terminatore.

Per ognuno dei char della stringa aggiungo alla variabile somma il valore del char stesso meno il valore del carattere '0'.

Ritorno la somma dei valori di tutte le cifre.

Continue

- Perché sottraggo ai caratteri della stringa il valore di '0'?
- È un modo comune per determinare il valore di un carattere che rappresenta una cifra data la sua rappresentazione ASCII:

Rappr. ASCII	Valore
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

- Se supponiamo che il carattere che stiamo esaminando abbia come rappresentazione nella tabella ascii il '6', il suo valore effettivo in memoria sarà 54.
- Sottraendo a questo carattere il valore effettivo della rappresentazione ascii di '0' sto in realtà facendo $54 - 48 = 6$!
- In questo modo rimuovo l'*offset* che c'è nella tabella ASCII tra i veri valori delle cifre e i valori delle loro rappresentazioni ASCII.

Continue

- Anche in questo caso, il problema è evidente, se uno dei caratteri presenti nella stringa non è una cifra lo aggiungiamo lo stesso a `somma`!
- Anche questa volta ci potremmo arrangiare con un goto modificando in questo modo la funzione:

```
size_t somma_cifre_dec(const char *s) {  
    size_t i, somma = 0;  
  
    for (i = 0; s[i] != 0; ++i) {  
        if (!isdigit(s[i]))  
            goto ricomincia_ciclo;  
  
        somma += s[i] - '0';  
  
        ricomincia_ciclo: ;  
    }  
  
    return somma;  
}
```

Uso la funzione `isdigit()` che trovo in `ctype.h` per determinare se il carattere corrente è una cifra.

Aggiungo il goto per saltare alla fine del ciclo e una label apposita.

Saltare alla label `ricomincia_ciclo` porta ad eseguire un comando vuoto alla fine del ciclo, per questo è necessario un punto e virgola da solo.

Continue

- Anche in questo caso in C esiste anche un altro comando, che ha un comportamento simile a quello del break (nel senso che sostituisce un goto), per fare questa cosa.
- Il comando in questione si chiama *continue*.
- Come per il break la sua sintassi è data solamente dalla keyword stessa:

`continue ;`

- Quando durante l'esecuzione di un ciclo viene incontrato un continue, tutti i comando successivi all'interno del corpo del ciclo non vengono eseguiti e l'esecuzione riparte dall'iterazione successiva.

Continue

- Utilizzando il continue la funzione che somma le cifre diventerà quindi:

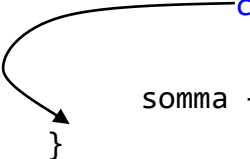
```
size_t somma_cifre_dec(const char *s) {  
    size_t i, somma = 0;  
  
    for (i = 0; s[i] != 0; ++i) {  
        if (!isdigit(s[i]))  
            continue;  
  
        somma += s[i] - '0';  
    }  
  
    return somma;  
}
```

- Anche qui il continue ci permette di ottenere lo stesso risultato in modo molto più chiaro ed elegante.

Continue

- In questo caso, se si verificasse la condizione dell'if allora eseguiremmo il comando continue e il ciclo ricomincerebbe.

```
size_t somma_cifre_dec(const char *s) {  
    size_t i, somma = 0;  
  
    for (i = 0; s[i] != 0; ++i) {  
        if (!isdigit(s[i]))  
            continue;  
        somma += s[i] - '0';  
    }  
  
    return somma;  
}
```




- Attenzione, in questo caso il `++i` che normalmente sarebbe stato eseguito alla fine del for viene eseguito lo stesso!

Continue

- Attenzione però ad utilizzare correttamente il continue, se ad esempio riadattassimo la funzione per utilizzare un while al posto del for avremmo qualcosa del genere:

```
size_t somma_cifre_dec(const char *s) {  
    size_t i = 0, somma = 0;  
  
    while (s[i] != 0) {  
        if (!isdigit(s[i]))  
            continue;  
  
        somma += s[i] - '0';  
        ++i;  
    }  
  
    return somma;  
}
```

Errore! Il ++i non viene più eseguito se si incontra qualcosa che non è un numero!



- Qui avremmo saltato anche il ++i e avremmo avuto un ciclo infinito non appena la funzione non avesse incontrato un carattere diverso dalle cifre decimali!

Switch

- Supponiamo ora di dover scrivere una funzione che esegue una certa operazione tra due numeri in base a un parametro che le viene passato.
- La sua dichiarazione potrebbe essere la seguente:

```
extern double esegui_operazione(char op, double n1, double n2);
```

- Il parametro `op` contiene l'operatore da applicare tra i due numeri `n1` e `n2`.
- Supponiamo per ora che le operazioni ammesse siano somma, sottrazione, moltiplicazione e divisione. Il parametro `op` conterrà quindi i caratteri `'+'`, `'-'`, `'*'` e `'/'`.
- La funzione ritorna poi il risultato in un `double`.

Switch

- la definizione della funzione potrebbe essere scritta così:

```
double esegui_operazione(char op, double n1, double n2) {  
    double risultato;  
  
    if (op == '+')  
        risultato = n1 + n2;  
    else if (op == '-')  
        risultato = n1 - n2;  
    else if (op == '*')  
        risultato = n1 * n2;  
    else if (op == '/')  
        risultato = n1 / n2;  
    else  
        risultato = 0;  
  
    return risultato;  
}
```

- Questa prolissa serie di if e else if serve solamente a confrontare il valore del parametro op con delle costanti.
- Si poteva scrivere meglio?

Switch

- La risposta è sì, in C esiste il comando *switch* che serve esattamente a questo.
- La sua sintassi è:

```
switch ( <expression> ) <comando>
```

- La *expression* tra parentesi tonde è **un'espressione di tipo intero** da valutare.
- Il *comando* contiene delle label particolari con la sintassi seguente:

```
case <const-expression> : <comando>
```

```
default : <comando>
```

- In questo caso *const-expression* deve essere un'espressione intera costante valutabile a tempo di compilazione.

Switch

- In questo caso la funzione di prima si potrebbe modificare ottenendo il codice seguente:

```
double esegui_operazione(char op, double n1, double n2) {  
    double risultato;  
  
    switch (op) {  
        case '+': risultato = n1 + n2;  
            break;  
        case '-': risultato = n1 - n2;  
            break;  
        case '*': risultato = n1 * n2;  
            break;  
        case '/': risultato = n1 / n2;  
            break;  
        default: risultato = 0;  
            break;  
    }  
  
    return risultato;  
}
```

- Nel nostro caso l'espressione intera da valutare è semplicemente `op`, a seconda del suo valore dovremo saltare a un caso diverso.

Switch

- Una volta valutata l'espressione tra le parentesi tonde dello switch l'esecuzione salta direttamente alla label che dopo la keyword case contiene l'espressione costante uguale a quella valutata.
- Nel caso non ci sia nessun case con un'espressione uguale allora l'esecuzione salta alla label default.
- Il comando switch è l'unico altro caso in cui si può utilizzare il break. In questo caso non interrompe un ciclo ma salta alla fine dello switch.
- Se non ci fossero i break alla fine di ogni caso l'esecuzione continuerebbe eseguendo anche il codice relativo agli altri casi!

Switch

- Supponiamo di chiamare la funzione con i seguenti parametri:

```
double ris = esegui_operazione('-', 34.3, 76.6);
```

- L'esecuzione seguirà quindi il seguente percorso:

```
double esegui_operazione(char op, double n1, double n2) {  
    double risultato;  
    switch (op) {  
        case '+': risultato = n1 + n2;  
            break;  
        case '-': risultato = n1 - n2;  
            break;  
        case '*': risultato = n1 * n2;  
            break;  
        case '/': risultato = n1 / n2;  
            break;  
        default: risultato = 0;  
            break;  
    }  
    return risultato;  
}
```

Viene valutata l'espressione `op` che in questo caso vale `' - '`.

L'esecuzione si sposta all'etichetta corrispondente al `' - '`.

Quando viene incontrato il `break` l'esecuzione salta alla prima espressione dopo la fine dello switch.

Switch

- A volte può capitare di eseguire parecchie istruzioni e di aver bisogno di variabili temporanee differenti all'interno dei singoli casi dello switch.
- In tal caso bisogna utilizzare le parentesi graffe per creare uno scope apposta per il singolo caso:

```
...  
case '*': {  
    int tmp = 0;  
    // altre operazioni...  
    break;  
}  
...
```

- In questo modo ogni case ha il suo scope con le proprie variabili locali.
- In generale lo switch non è un costrutto fondamentale, tutto ciò che si fa con uno switch si può fare benissimo con una serie di if ed else. Il suo utilizzo, in alcuni casi, rende solamente il codice leggermente più elegante e leggibile.

Enum

- In C è stata introdotta una keyword per definire degli insiemi di costanti intere.
- Questa keyword è la parola **enum**.
- Un enum (o enumerazione) ci permette di definire una serie di costanti intere permettendoci di specificare o meno il loro valore.
- La sintassi è la seguente:

```
enum <identifier> { <enumerator-list> } ;
```

- Dove *identifier* è il nome che scegliamo di dare a questo insieme di costanti e *enumerator-list* è la lista delle costanti che vogliamo definire.
- La *enumerator-list* è una lista di nomi separati da virgole in cui opzionalmente, attraverso l'uguale, si può anche stabilire il valore di una costante.

Enum

- La sintassi per definire un elemento della lista è la seguente:

<identifier> oppure

<identifier> = <constant-expression>

- I valori delle costanti dichiarate nella lista partono da 0 e per ogni nuova costante il valore si incrementa di 1 rispetto a quella precedente.
- Questo resta valido anche quando il valore di una costante viene specificato manualmente. I valori delle successive proseguiranno da quel valore incrementandolo di 1.
- I valori che si attribuiscono alle costanti possono essere solamente delle espressioni intere costanti.

Enum

- In questo esempio dichiaro un enum chiamata `colori` che contiene cinque costanti.

```
enum colori {  
    rosso,  
    blu,  
    verde,  
    bianco,  
    nero  
};
```

In questo caso non sono stati specificati valori per le costanti quindi il compilatore assegnerà ad esse dei valori incrementati di uno ogni volta a partire da 0:

- `rosso = 0`
- `blu = 1`
- `verde = 2`
- `bianco = 3`
- `nero = 4`

Enum

- Aggiungiamo ora alcuni valori specificati manualmente da noi:

```
enum colori {  
    rosso,  
    blu = 18,  
    verde,  
    bianco = 32,  
    nero  
};
```

Questa volta ho specificato il valore della costante `verde` quindi tutte quelle successive avranno valori ogni volta incrementati di 1 a partire da quello di `verde`:

- `rosso = 0`
- `blu = 18`
- `verde = 19`
- `bianco = 32`
- `nero = 33`

Enum

- Se è necessario è possibile anche specificare manualmente i valori di tutte le costanti:

```
enum colori {  
    rosso = 4,  
    blu = 3,  
    verde = 18,  
    bianco = 9,  
    nero = 1  
};
```

In questo caso ho specificato manualmente tutti i valori delle costanti, di conseguenza il compilatore non assegna nessun valore diverso ad esse:

- rosso = 4
- blu = 3
- verde = 18
- bianco = 9
- nero = 1

- Nota: è anche possibile specificare che due costanti abbiano lo stesso valore, potrei ad esempio assegnare sia a `rosso` che `blu` il valore 4 senza problemi.

Enum

- Attenzione ad usare gli enum.
- A differenza di quanto visto con le struct, il nome dell'enumerazione finisce nello namespace dei tag delle struct ma tutti i nomi che dichiariamo all'interno di un enum finiscono nel namespace globale!
- Questo significa che una volta dichiarato l'enum non sarà più possibile definire variabili o funzioni con nomi che compaiono tra quelli delle costanti dell'enum!
- Ad esempio, il seguente codice non compilerebbe:

```
int main(void) {  
    enum colori {  
        rosso,  
        blu,  
        verde,  
        bianco,  
        nero  
    };  
  
    int blu = 32;  
  
    return 0;  
}
```

Se provassi a compilare il compilatore mi darebbe un errore a questa riga, dicendomi che sto provando a ridefinire `blu` che era già stata definita in precedenza

Enum

- A questo punto definire delle costanti usando un enum oppure definendo delle variabili globali costanti non è poi molto diverso...
- Uno dei vantaggi potrebbe essere quello di riuscire a tenere meglio organizzate le costanti nel proprio programma.
- È anche possibile utilizzare un enum come tipo di dato per dichiarare parametri o variabili, ad esempio:

```
enum oper { add, sub, mul, div };
```

```
double op(enum oper op, double n1, double n2) {  
    double risultato;  
    switch (op) {  
        case add: risultato = n1 + n2; break;  
        case sub: risultato = n1 - n2; break;  
        case mul: risultato = n1 * n2; break;  
        case div: risultato = n1 / n2; break;  
        default: risultato = 0; break;  
    }  
    return risultato;  
}
```

```
int main(void) {  
    double ris = op(add, 34.3, 76.6);  
    return 0;  
}
```

Union

- In C è possibile dichiarare anche un altro tipo di dato strutturato.
- Il loro nome è **union** e la sintassi per la dichiarazione è praticamente identica a quella che abbiamo già visto per le struct:

```
union <tag> { <declaration-list> } ;
```

- La differenza però è sostanziale.
- Se nelle struct il compilatore si preoccupa di mantenere in memoria lo spazio necessario per mantenere *tutte* le variabili contemporaneamente, nelle union viene allocato solamente lo spazio per contenere la variabile più grande e tale spazio viene usato per tutte le variabili definite nella union.
- Si può pensare a una union come a una struct in cui tutte le variabili sono «sovrapposte» in memoria allo stesso indirizzo.

Union

- Facciamo un esempio per capire meglio il funzionamento di una union confrontato a quello di una struct.
- Consideriamo due dichiarazioni simili, prima di una struct e poi di una union:

```
struct esempio_s {  
    char c;  
    short s;  
    int i;  
};
```

```
union esempio_u {  
    char c;  
    short s;  
    int i;  
};
```

- In queste due dichiarazioni abbiamo inserito sia nella struct che nella union gli stessi membri.
- Utilizziamo l'operatore sizeof su entrambi per verificarne il risultato:

```
size_t s1, s2;  
s1 = sizeof(struct esempio_s);  
s2 = sizeof(union esempio_u);
```


Union

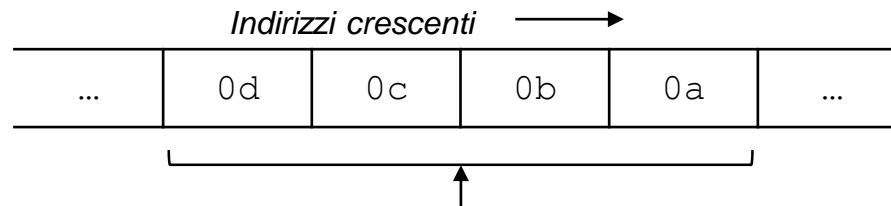
- Esaminiamo entrambi i casi:
 1. Nel caso della struct sappiamo che il compilatore riserva per le variabili di questo tipo *almeno* i byte sufficienti per memorizzare tutti i membri. Visto che la struct contiene un char, uno short e un int ci aspettiamo che la dimensione sia almeno $1 + 2 + 4 = 7$ byte. Eseguendo il codice scopriremmo che `s1` vale 8, quindi possiamo memorizzare tutti gli elementi. Evidentemente il compilatore ha ritenuto opportuno aggiungere un byte in più per questioni di efficienza.
 2. Nel caso della union abbiamo detto che il compilatore riserva spazio solo per contenere la variabile di dimensione maggiore, nel nostro caso un int. Effettivamente `s2` vale proprio 4, cioè la dimensione di un int.
- Come funziona quindi l'accesso ai membri di una union se non c'è abbastanza spazio in memoria per contenerli tutti ed essi sono sovrapposti?

Union

- Accedendo ai membri di una union in realtà accediamo sempre allo stesso indirizzo in memoria.
- Ogni accesso prova a leggere da quell'indirizzo un variabile del tipo del membro a cui abbiamo provato ad accedere!
- Supponiamo di eseguire il seguente codice:

```
union esempio_u a;  
a.i = 0x0a0b0c0d;
```

- Abbiamo definito una variabile `a` di tipo `union esempio_u` e abbiamo assegnato al suo membro `i` (quello di tipo `int`) il valore `0x0a0b0c0d`.
- La situazione in memoria sarà la seguente:



- Abbiamo riempito tutti e 4 i byte della union con il valore dell'intero `a.i` (assumiamo che l'intero sia memorizzato in little-endian).

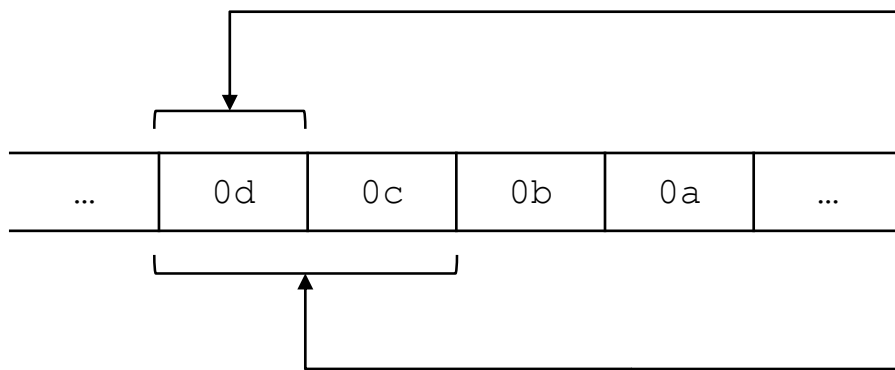
Union

- Cosa succederebbe se ora provassimo ad accedere ad `a.c` o ad `a.s`? Cosa otterremmo dopo aver eseguito il seguente codice?

```
char c = a.c;
```

```
short s = a.s;
```

- Scopriremmo che `c` e `s` valgono rispettivamente `0x0d` e `0x0c0d`!
- Questo accade perché accedendo ad `a.c` e ad `a.s` accediamo alla stessa locazione in memoria di `a.i` ma questa volta proviamo a leggere prima un char poi uno short!



Con il primo accesso leggiamo **un byte** perché proviamo a leggere un char.

Con il secondo accesso invece leggiamo **due byte** perché proviamo a leggere uno short.

Union

- L'inizializzazione di una union avviene allo stesso modo delle struct, la sintassi è esattamente la stessa, ad esempio:

```
union esempio_u b = { 0xff };
```

```
union esempio_u c = { .c = 0x0a, .s = 0x0c0d, .i = 0xffff };
```

- Se non viene specificato il membro, come nel primo caso, conta l'ordine, il valore `0xff` verrà usato per inizializzare il primo membro, ossia `b.c`.
- Ovviamente bisogna però prestare attenzione nel secondo caso.
- L'inizializzazione di `c.c` e `c.s` sarà totalmente inutile perché la memoria riservata per la union verrà riempita completamente dall'inizializzazione di `c.i`.

Union

- Il motivo per cui le union esistono è principalmente quello di risparmiare memoria.
- Poter memorizzare nella stessa locazione di memoria dati di tipo diverso ci permette di risparmiare qualche byte.
- Bisogna però ovviamente avere la certezza che non sia mai necessario memorizzare dati in più di un membro della union per volta.
- Attenzione però, non c'è alcun modo di verificare quale sia il membro attivo in una union in un certo momento!
- Leggere un membro non attivo da una union inconsapevolmente può portare rapidamente un programma ad avere un comportamento difficilmente prevedibile.
- Quando si usano delle union bisogna prevedere attentamente tutti i casi in cui viene effettuato l'accesso ai suoi membri per evitare di introdurre bug nel proprio codice.