Lezione 3

Procedura soluzione problemi
Assegnamento
Compendio sintassi
Tipo booleano ed operatori logici
Espressioni aritmetiche e logiche

Soluzione problemi

- Data la centralità di questo aspetto per la vostra professionalità
  - Nella prossima slide rivediamo la procedura corretta per passare da un problema ad una soluzione valida

Sviluppo di una soluzione

- Un buon ordine con cui arrivare a risolvere, mediante un programma, un problema nuovo di cui non si conosce la soluzione è il seguente:

1) Riflettere sul problema finché non si è sicuri di aver capito a sufficienza tutti gli aspetti e le implicazioni
2) Cercare di farsi venire un'idea che sembri buona per risolvere il problema (o almeno per partire)
3) Provare a definire l'algoritmo e controllarlo per capire se è corretto (eventualmente modificarlo)
4) Quando si è sicuri dell'algoritmo, partire con la codifica
5) Collaudare il programma per verificare che faccia veramente quello che deve

Assegnamento

Istruzione di assegnamento

- Espressione di assegnamento:

  nome_variabile **=** *<espressione>*

- Istruzione di assegnamento:

  <espressione di assegnamento> ***;***

  - E' cioè una espressione di assegnamento **seguita da un *;***
  - Viene utilizzata per assegnare ad una variabile (non ad una costante!) il valore di un'espressione

Domanda

- Quali operazioni bisogna effettuare sulla memoria per assegnare un nuovo valore ad una variabile?

Risposta

- Bisogna scrivere il nuovo valore all'interno delle celle di memoria in cui è memorizzata la variabile
- Più a basso livello: bisogna scrivere una nuova configurazione di *bit*
  - quella che rappresenta il nuovo valore

Assegnamento e memoria

Esempio

```
int  N=10;
```

simbolo	*indirizzo*
N	**1600**

1600

...
10
...

L'esecuzione di una **definizione** provoca l'allocazione di uno spazio in memoria pari a quello necessario a contenere un dato del tipo specificato

```
N = 150;
```

simbolo	*indirizzo*
N	**1600**

1600

...
150
...

L'esecuzione di un **assegnamento** provoca l'inserimento nello spazio relativo alla variabile del valore indicato a destra del simbolo **=**

Domanda

- Quale informazione bisogna avere per poter modificare il valore di una variabile in memoria?

Risposta

- Bisogna sapere dove si trova in memoria!
- Occorre sapere cioè il suo **indirizzo**
  - Ossia l'indirizzo della prima delle celle consecutive in cui è memorizzata la variabile

lvalue e rvalue

- Come si effettua quindi l'assegnamento? Consideriamo, per esempio, il precedente assegnamento:
  `N = 150;`
- Viene preso l'indirizzo della variabile individuata dall'identificatore a sinistra dell'assegnamento (l'identificatore è **`N`** nel nostro esempio)
  - tale indirizzo è detto **lvalue** (left value)
- Viene calcolato il valore dell'espressione che compare a destra (150 nell'esempio) ed assegnato all'oggetto memorizzato all'indirizzo (lvalue) ottenuto nel passo precedente
  - tale valore è detto **rvalue** (right value)

Ordine di esecuzione

- L'esecuzione di un'istruzione di assegnamento comporta **prima** la valutazione di tutta l'espressione a destra dell'assegnamento.
  Esempi:
  ```
  int c, d;
  c = 2;
  d = (c+5)/3 - c;
  d = (d+c)/2;
  ```
- Solo **dopo** si inserisce il valore risultante (*rvalue*) nella spazio di memoria dedicato alla variabile

Risultato assegnamento 1/2

- Come tutte le espressioni, anche l'espressione di assegnamento ha un proprio valore
- In particolare ha per valore *l'indirizzo della variabile a cui si è assegnato il nuovo valore* (quindi l'*lvalue*)
  Esempio: l'espressione
  `a = 3`
  ha per valore l'indirizzo di **`a`**
- Uno dei modi in cui si può sfruttare tale indirizzo è per effettuare *assegnamenti multipli*, ad esempio:

```
int c, d;
c = d = 2;
```

  - L'effetto della seconda istruzione, che, come si vedrà meglio in seguito, è equivalente a
    `c = (d = 2) ;`
    è il seguente:

Risultato assegnamento 2/2

- L'espressione **`d = 2`** produce come valore l'indirizzo della variabile **`d`**
- L'espressione **`c =`** ... si aspetta a destra un valore da assegnare a **`c`**
  - Siccome si ritrova invece l'indirizzo di una variabile, tale indirizzo viene utilizzato per accedere al (nuovo) valore della variabile **`d`** (ossia 2), ed utilizzarlo per assegnare il nuovo valore a **`c`**
- In definitiva dopo l'istruzione
  `c = (d = 2) ;`
  sia **`c`** che **`d`** hanno il valore 2

Stato conoscenze

- A questo punto conosciamo due istruzioni del linguaggio
  - Definizione
  - Assegnamento
- Inoltre abbiamo imparato un po' ad usare gli oggetti *cout* e *cin* senza indagare sul meccanismo interno con cui funzionano

Esercizio: numero al contrario

- Specifica del problema
  - come sempre accadrà d'ora in poi, nel nostro caso la specifica è una semplice traccia:

- Leggere da *stdin* un numero intero positivo, che si assume essere compreso tra 100 e 999 (lo si dà per scontato senza effettuare controlli), e stamparlo al contrario (con le cifre in ordine inverso)

- Esempi:
  103 → 301
  230 → 032
  527 → 725

Procediamo con ordine

- Questo è un problema per risolvere il quale dobbiamo riflettere bene ...
- Per fare un buon lavoro, rispettiamo le fasi di sviluppo riviste all'inizio di questa lezione
- Quindi analizziamo prima di tutto con calma il problema
- Solo dopo di essere sicuri di aver chiaro il problema anche nei dettagli, cerchiamo di farci venire un'idea (chiara) su come risolverlo
- Se non arrivano idee, c'è un suggerimento nella prossima slide ...

Suggerimento 1/3

- Prendiamo un numero qualsiasi, per esempio 573
- Quanto vale 573 % 10 ?

Suggerimento 2/3

- Vale 3
- Ossia proprio il valore della sola cifra delle unità ...
- E se volevamo ottenere il valore della cifra delle decine?
  - Purtroppo 573%100 non è uguale alla cifra delle decine, cioè 7, ma è uguale a 73
  - Per poter estrarre le decine, potrei utilizzare %10 se riuscissi prima a trasformare 573 in 57 ...
  - Come potrei fare?

Suggerimento 3/3

- Lo divido semplicemente per 10 (divisione intera)
  573 / 10 = 57
- Quindi quanto farà (573/10) % 10 ?
- Una volta capito come ottenere anche le decine, dovreste essere pronti per l'idea completa

Idea!

- Utilizzare le operazioni di modulo e di divisione fra numeri interi
- Dato un numero, valgono le seguenti relazioni:
  - Unità = numero%10;
    - Esempio: 234%10 = 4
  - Decine = (numero/10^1)%10;
    - Esempio: (234/10)%10 = 3
  - Centinaia = (numero/10^2)%10;
    - Esempio: (234/100)%10 = 2

Algoritmo

- Dato il numero letto da *stdin* (ad esempio 234)
  - Memorizzare in una variabile **`unita`** il risultato di **`numero % 10`** (che ci restituisce proprio le unità)
    Nel nostro esempio otteniamo: **`unita = 4`**
  - Memorizzare in una variabile decine il risultato di **`(numero/10) % 10`**
    Nel nostro esempio: **`decine = 23%10 = 3`**
  - Memorizzare in una variabile **`centinaia`** il risultato di **`(numero/100) % 10`**
    Nel nostro esempio: **`centinaia = 2%10 = 2`**

Programma

```
main()
{
  int numero;
  int unita, decine, centinaia ;

  cin>>numero;

  unita = numero % 10;
  decine = (numero/10)%10;
  centinaia = (numero/100)%10;

  cout<<unita<<decine<<centinaia<<endl;
}
```

Esercizio per casa

- Leggere da *stdin* un numero intero positivo, che si assume non essere multiplo di 10 ed essere compreso tra 101 e 999 (senza effettuare controlli), e **memorizzare in una variabile intera** un numero intero le cui cifre siano in ordine inverso rispetto al numero letto da *stdin*; stampare infine il numero ottenuto

- Esempi:
  103 → 301
  234 → 432
  527 → 725

- Idea, algoritmo e soluzione nelle prossime due slide

Idea

- In aggiunta all'idea di base già vista per l'esercizio precedente, nella variabile intera in cui va memorizzato il numero al contrario, la cifra memorizzata dentro **`unita`** deve indicare le centinaia, quindi va moltiplicata per 100
  Nel nostro esempio, 4 deve diventare 400
- La cifra memorizzata dentro **`decine`** deve indicare le decine, quindi va moltiplicata per 10
  Nel nostro esempio, 3 deve diventare 30
- La cifra memorizzata dentro **`centinaia`** deve indicare le unità, quindi non va moltiplicata per nulla
  Nel nostro esempio, 2 deve rimanere 2

Algoritmo

- Si può esprimere l'algoritmo con una semplice formula algebrica
- Il numero da memorizzare nella variabile intera andrà calcolato come:

```
unita*100 + decine*10 + centinaia
```

Soluzione

```
main()
{
  int numero;
  int unita, decine, centinaia, risultato;

  cin>>numero;

  unita = numero % 10;
  decine = (numero/10)%10;
  centinaia = (numero/100)%10;

  risultato = unità*100 + decine*10 + centinaia;
  cout<<risultato<<endl;
}
```

Sintasssi del C/C++

Altre note sulla sintassi

- Nella definizione della sintassi si utilizza spesso la notazione
  <*elemento_linguistico*> ::= …
  - Il significato è che l'elemento linguistico è definito, sintatticamente, nel modo descritto a destra del simbolo ::=
- Le parentesi graffe **{** <costrutto> **}** indicano la possibilità che un certo costrutto sia ripetuto 0 o più volte
  - Ossia che non ci sia affatto, oppure che appaia una o più volte

Sintassi del C/C++ 1/3

- Ora che abbiamo più familiarità col linguaggio, fissiamo un po' meglio la sintassi ...
- Un programma C/C++ è una sequenza di parole (**token**) delimitate da spazi bianchi (**whitespace**)
  - *Spazio bianco*: carattere spazio, tabulazione, a capo
  - *Parola*: sequenza di lettere o cifre non separate da spazi bianchi

Token possibili: *operatori, separatori, identificatori, parole chiave (riservate), commenti, espressioni letterali*

  - Operatore: denota una operazione nel calcolo delle espressioni
  - Separatore: `( ) , ; : { }`

Sintassi del C/C++ 2/3

IDENTIFICATORI

<Identificatore> ::= *<Lettera>* **{** *<Lettera>* | *<Cifra>* **}**

- *<Lettera>* include tutte le lettere, maiuscole e minuscole, e l'underscore "_"
- La notazione **{** A | B **}** indica una sequenza indefinita di elementi A o B
- Maiuscole e minuscole sono considerate **diverse** (il linguaggio C/C++ è *case-sensitive*)

- Informalmente: una lettera seguita da 0 o più lettere o cifre

Esempi di identificatori corretti

```
a
b2
pippo3rw
```

Esempi errati

```
2
2ert
```

Sintassi del C/C++ 3/3

PAROLE CHIAVE (RISERVATE)

- **`int, float, double, char, if, for, do, while, switch, break, continue, …`**
  `{ }` delimitatore di blocco

COMMENTI

- **`// commento, su una sola riga`**
- **`/* commento,`**
  `anche su più righe  */`
  - Una delle formattazioni tipiche è

```
/*
 * Qui il commento
 * di più righe
 */
```

Uso degli spazi bianchi

- Una parola chiave ed un identificatore **vanno separati da almeno uno spazio bianco**
- Esempio:
```
int a; // inta sarebbe un identificatore !
```
- In tutti gli altri casi gli spazi bianchi <u>non sono obbligatori</u>
  - Li si utilizza però per migliorare la leggibilità del programma per un 'umano'
- Si può separare una coppia di *token* consecutivi col <u>numero ed il tipo di spazi bianchi che si preferisce</u> (ripetiamo che va messo almeno uno spazio bianco solo nel caso si tratti di una parola chiave seguita da un identificatore)

Tipo booleano

Tipo booleano

- Disponibile in C++, ma non in C
- Nome del tipo: **`bool`**
- Valori possibili: vero (**true**), falso (**false**)
  - Identificati dai due letterali booleani:
    `true`
    `false`

```
Creating test database for alias 'default'...
F.
======================================================================
FAIL: test_home_page_returns_correct_html (lists.tests.HomePageTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/max4/Data/Programming/TDD/superlists/lists/tests.py",
rns_correct_html
    self.assertTrue(response.content.startswith(b'<html>'))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Definizione

- Solita sintassi per le definizioni
- Esempio:

```
bool u, v = true ;  // la seconda variabile
                    // è inizializzata a vero
```

- Operazioni possibili: ...

Operatori logici: sintassi

operatore logico	*numero argomenti*	**sintassi (posizione)**	**esempi**
not logico (negazione)	*uno (unario)*	**!** (prefisso)	`bool b, a = !true ;` `b = !a ;`

and logico (congiunzione)	*due (binario)*	**&&** (infisso)	`bool b, a, c ;`
			`c = a && b ;`
			`b = true && a ;`

or logico (disgiunzione)	*due (binario)*	**||** (infisso)	`bool b, a, c ;`
			`c = a || b ;`
			`b = true || a ;`

Che valori ritornano questi operatori?

La loro semantica è definita dalle cosiddette *tabelle di verità*

Tabella di verità

AND	*Ris.*	OR	*Ris.*	NOT	*Ris.*
V && V	V	V || V	V	!V	F
V && F	F	V || F	V	!F	V
F && V	F	F || V	V		
F && F	F	F || F	F		

Domanda

- Che valore ha una variabile booleana non inizializzata?

Risposta

- Per il momento assumiamo che sia casuale
  - Potrebbe tanto vero quanto falso

Tipo booleano e tipi numerici

- Se un oggetto di tipo booleano è usato dove è atteso un valore numerico
  - **`true`** è convertito a 1
  - **`false`** è convertito a 0
- Viceversa, se un oggetto di tipo numerabile è utilizzato dove è atteso un booleano
  - ogni valore diverso da 0 è convertito a **`true`**
  - il valore 0 è convertito a **`false`**

Esercizio

- *stampa_bool.cc* della terza esercitazione

Tipo booleano e linguaggio C

- In C, non esistendo il tipo `bool`, gli operatori logici
  - operano su interi
    - il valore 0 viene considerato falso
    - ogni valore diverso da 0 viene considerato vero
  - e restituiscono un intero:
    - il risultato è 0 o 1
- Esempi di espressioni con operatori logici (che in C++ ritornerebbero **true** o **false**)

```
5 && 7          0 || 33          !5
```

Booleani e valori interi in C++

- Anche in C++ si possono utilizzare gli interi dove sono attesi dei booleani, esattamente come in C
- Tali valori sono convertiti nel modo seguente:
  - il valore 0 viene convertito a **false**
  - ogni valore diverso da 0 viene convertito a **true**
- Ovviamente utilizzare i booleani al posto di *sovraccaricare* il significato degli interi
  - Rende i programmi molto più chiari
  - E' esattamente il motivo per cui sono stati introdotti i booleani

Operatori di confronto

Operatori di confronto 1/2

`==` Operatore di confronto di uguaglianza
(il simbolo = denota invece l'operazione di assegnamento!)

`!=` Operatore di confronto di diversità

`>` Operatore di confronto di maggiore stretto

`<` Operatore di confronto di minore stretto

`>=` Operatore di confronto di maggiore-uguale

`<=` Operatore di confronto di minore-uguale

- Restituiscono un valore di tipo **booleano**: **`true`** oppure **`false`**

Operatori di confronto 2/2

- Gli operatori di confronto si possono applicare sia agli oggetti di tipo **int** che agli oggeti di tipo **bool**
  - E, come vedremo in seguito, anche ad altri tipi di oggetti

Esercizio

- *stampa_logica_semplice.cc* della terza esercitazione

Gruppi facebook

facebook.com/groups/informaticaunimore2012

facebook.com/groups/informatica.unimore.2013

facebook.com/groups/informatica.unimore.1415

Espressioni

Espressioni

- Costrutto sintattico formato da letterali, identificatori, operatori, parentesi tonde, ...
- Operatori
  - Moltiplicativi: *   /   %
  - Additivi: +   -
  - Traslazione: <<   >>   (Programmazione II)
  - Relazione (confronto): <   >   <=   >=
  - Eguaglianza (confronto): ==   !=
  - Logici: !   && || (ce ne sono anche altri)
  - Assegnamento: =   +=   -=   *=   /=
- Abbiamo già visto quasi tutti questi operatori parlando del tipo **int** e del tipo **bool**

Domanda

- Data una variabile booleana **`x`**, che differenza c'è tra i valori delle espressioni per ciascuna delle seguenti coppie?

```
x == true          x

x == false         !x
```

Risposta

- **Nessuna**
- Un programmatore utilizza sempre la forma sintatticamente e concettualmente più semplice per una data espressione
- Quindi nelle precedenti coppie di espressioni sono da preferire:

```
x

!x
```

Altri operatori

- Assegnamento abbreviato: +=, -=, *=, /=, ...
  `a += b ;    ↔    a = a + b ;`

- Incremento e decremento: `++`   `--`

  - Prefisso: prima si effettua l'incremento/decremento, poi si usa la variabile. Restituisce un **lvalue** (l'indirizzo della variabile incrementata)

```
int a = 3; cout<<++a; // stampa 4
(++a) = 4; // valido, cosa assegna ad a?
```

  - Postfisso: prima si usa il valore della variabile, poi si effettua l'incremento/decremento. Restituisce un **rvalue**

```
int a = 3; cout<<a++; // stampa 3
cout<<a; // stampa 4
(a++) = 7; // ERRORE !!!
```

Tipi di espressioni

- Un'espressione si definisce
  - aritmetica: produce un risultato di tipo aritmetico
  - logica: produce un risultato di tipo booleano
- Esempi:

Espressioni aritmetiche

```
2 + 3

( 2 + 3 ) * 5
```

```
4 > 2

true || (2 > 5)
```

Espressioni logiche

Proprietà degli operatori 1/2

- **Posizione** rispetto ai suoi operandi (o argomenti): prefisso, postfisso, infisso
- **Numero di operandi (arietà)**
  - Unari se hanno un solo operando
    - Esempi: `!a` `++a` `a--`
  - Binari se hanno due operandi
    - Esempi: `a && b` `a + b`
  - Ternari
    - Vedremo un esempio in seguito

Proprietà degli operatori 2/2

- **Precedenza** (o **priorità**) nell'ordine di esecuzione
  - Es: `1 + 2 * 3` è valutato come `1 + (2 * 3)`
    `k<b+3` è valutato come `k<(b+3)`, e non `(k<b) +3`
- **Associatività**: ordine con cui vengono valutati due operatori con la stessa precedenza.
  - Associativi a sinistra: valutati da sinistra a destra
    - Es: `/` è associativo a sinistra, quindi `6/3/2` è uguale a `(6/3)/2`
  - Associativi a destra: valutati da destra a sinistra
    - Es: `=` è associativo a destra ...

Associatività assegnamento

- L'operatore di assegnamento può comparire più volte in un'istruzione.

- L'<u>associatività</u> dell'operatore di assegnamento è a **destra**
  Esempio:
  ```
  k = j = 5;
  ```
  equivale a `k = (j = 5);` ossia:
  ```
  j = 5;
  k = j;
  ```

- Invece:
  ```
  k = j + 2 = 5; // ERRORE !!!!!
  ```
  perché `j + 2` non può fornire un **lvalue**, ossia l'indirizzo di una variabile!

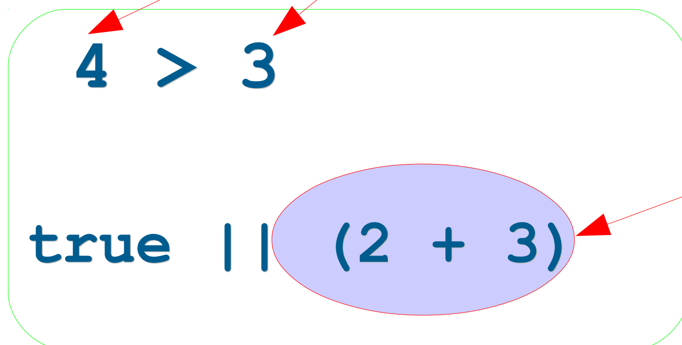
Ordine valutazione espressioni

- Si calcolano prima i fattori, quindi i termini
  - **Fattori**: ottenuti dalle espressioni letterali e dal calcolo delle funzioni e degli operatori unari
  - **Termini**: ottenuti dal calcolo degli operatori binari
    - Moltiplicativi: * / %
    - Additivi: + -
    - Traslazione: << >>
    - Relazione: < > <= >=
    - Eguaglianza: == !=
    - Logici: && ||
    - Assegnamento: = += -= *= /=
- Con le parentesi possiamo modificare l'ordine di valutazione dei termini

Esempi

Espressioni aritmetiche

Fattori



Termini

Espressioni logiche

Sintesi priorità degli operatori

Categoria	Operatori
Fattori	! ++ --
Termini	* / %
	+ -
	> >= < <=
	== !=
	&&
	||
	? :
Assegnamento	=

Priorità decrescente

Esercizi

- *stampa_logica_composta.cc* e *stampa_1_se_in_intervallo.cc* della terza esercitazione