

Programmare è un'arte

Si tratta di mettere insieme tanti piccoli elementi nel migliore dei modi possibile per ottenere un **risultato che soddisfi le specifiche**

Dato un problema, ci sono **virtualmente infiniti programmi** in grado di risolverlo (vedi parte 1 del corso)

- Sono tutti equivalenti?
- **Prima** di scrivere un programma occorre:
 - Aver compreso il problema in maniera approfondita
 - Determinare **precisamente un algoritmo** che possa portare ad una soluzione **efficiente**
- **Mentre** si scrive un programma è necessario:
 - Conoscere quali sono i mattoni disponibili
 - **Linguaggio di programmazione**
 - **Librerie**
 - ...
 - Saper applicare **buoni principi di programmazione**

1

Principi di programmazione

- **Principi di base** (ma che cosa significano?)
 - Efficienza
 - Modularità della soluzione
 - Ordine e leggibilità
- **Efficienza**: determinazione di un algoritmo che consumi poche risorse (in termini di uso di memoria e di tempo di CPU principalmente) e sua codifica efficace e “sostenibile” (minime risorse utilizzate, senza penalizzare modularità e leggibilità)

2

Modularità

- Problema complicato (lo scoprirete sempre più quando programming-in-the-large)
- Alla base di tutto sta ***l'impostazione della soluzione (top-down)***
 - Determinare precisamente l'algoritmo di soluzione
 - ***Dividerlo in sotto problemi***
 - Dividere in sotto-sotto problemi i sotto problemi
 - ...
 - Costruire le singole sotto-parti di soluzione in modo che possano essere riusabili in altri contesti
 - Integrare opportunamente le sotto-parti

3

Modularità

- La ***modularità/riusabilità*** è un concetto chiave
 - Si evita di reinventare la ruota tutte le volte
 - Si fa affidamento su ***librerie ampiamente diffuse***, quindi testate e ***affidabili***
 - Moduli scritti per utilizzo da parte di una comunità
- Come gradevole effetto collaterale, il codice così scritto è più leggibile
 - ...ci saranno adeguati esempi più avanti...

4

Ordine

- È fondamentale che un programma sia **leggibile**:
 - **Facile comprensione** del codice da parte di chi non l'ha scritto ma vorrà mantenerlo (il codice deve essere "autoesplicativo" → anche buon uso dei commenti)
 - **Chiara strutturazione**
- Per aumentare la leggibilità:
 - Regole di **naming**
 - Regole di **strutturazione** del codice

5

Regole di Naming

- **I nomi di variabili e funzioni DEVONO avere nomi autoesplicativi**
 - Variabili di nome `pippo`, `pluto`, `paperino`, `a23`, `kk1`, `pa3`, ecc. non dicono nulla su ciò che contengono
- Usare il *Camel Casing* (prima lettera minuscola) sia per le variabili, sia per le funzioni:
 - `rowIndex`, `columnIndex`, `colorConverter...`
 - `void swapValues(int &firstValue, int &secondValue);`
 - `void saveToFile(int buffer[], int bufferSize);`

6

Regole di Indentazione

Blocchi di codice innestati vanno **opportunamente indentati**

- Normalmente l'indentazione è automaticamente effettuata dall'editor (specializzato – non il notepad)

```
if (pippo > 17)
{ printf("Sei maggiorenne!");
} else
{ printf("Sei minorenni!"); }
```

Quale è più
chiaro?

```
if (age >= 18)
{
    printf("Sei maggiorenne!");
}
else
{
    printf("Sei minorenni!");
}
```

Regole di indentazione

Poche e semplici...

- **Parentesi graffe sempre a capo**
- **Contenuto delle parentesi graffe sempre indentato** (di un *tab*) rispetto alle parentesi stesse
- **Non più di uno statement** per linea
- Se sulle slide queste regole non sono rispettate... è solo perché gli esempi non sempre vogliono stare racchiusi in una sola slide...

Un po' di idee

- Problema di calcolo
 - Può essere risolto eseguendo una serie di operazioni in un ordine opportuno
- **Algoritmo**: procedimento di soluzione in termini di
 - Azioni che devono essere eseguite
 - Ordine in cui queste azioni devono essere eseguite
- Controllo del programma (o del flusso di esecuzione)
 - Specifica l'ordine con cui le azioni devono essere eseguite

9

Pseudocodice

- **Linguaggio informale che aiuta nello sviluppo e nella rappresentazione degli algoritmi**
- Simile al linguaggio di tutti i giorni
- Non comprensibile dai calcolatori
- Aiuta il programmatore a “visualizzare” il programma prima di scriverlo
 - Facile da convertire nel corrispondente programma C

10

Bohm and Jacopini

- ***Tutti i programmi possono essere scritti in termini di tre strutture di controllo***
 - Sequenza: nativa in C
 - Gli *statement* sono eseguiti per *default* in modo sequenziale
 - Strutture di selezione: C ne ha tre tipi
 - `if`, `if...else`, `switch`
 - Strutture di ripetizione: C ne ha tre tipi
 - `while`, `do...while`, `for`

11

Espressioni logiche

- Alla base di tutto ci sono le ***condizioni, rappresentate da espressioni logiche***
- Un'espressione logica è un'espressione che può essere valutata ***come "vero" oppure "falso"***
- In C i valori "vero" e "falso" sono rappresentati, rispettivamente, da un valore intero diverso da zero e da un valore intero uguale a zero

12

Operatori Relazionali

Operatore standard	Operatore C	Esempio di condizione C	Significato della condizione C
=	==	$x == y$	x è uguale a y
≠	!=	$x != y$	x è diverso da y
>	>	$x > y$	x è maggiore di y
<	<	$x < y$	x è minore di y
>=	>=	$x >= y$	x è maggiore o uguale a y
<=	<=	$x <= y$	x è minore o uguale a y

Operatori relazionali per costruire condizioni

13

Operatori Logici

- **&& (AND logico)**
 - Restituisce **true** se entrambe le condizioni sono **true**
- **|| (OR logico)**
 - Restituisce **true** se una delle due condizioni è **true**
- **! (NOT logico, negazione logica)**
 - Rovescia la verità/falsità della condizione
 - Operatore unario: ha un solo operando!

14

if...else

■ Struttura tipica

```
if (espressione logica)
{
    /* sequenza di istruzioni */
}
else
{
    /* sequenza di istruzioni */
}
```

15

Primo esempio: voto di un esame

- Richiedere in ingresso un voto (valore da 0 a 33)
- Se il valore è inferiore a 18, stampare “Bocciato”
- Se il valore è almeno 18, stampare “Promosso”
- Se il valore è superiore a 30, stampare “Promosso con Lode”

16

Primo esempio: voto di un esame

- **Definire una variabile** per inserire il voto
- Stampare un messaggio per richiedere l'inserimento del valore
- Leggere il valore inserito dall'utente e **inserirlo nella variabile**
- Verificare il valore inserito
 - Se è minore di 18 → Bocciato
 - Altrimenti
 - Promosso
 - Se è maggiore di 30 → Lode!!!

17

Primo esempio: voto di un esame

```
#include <stdio.h>
int main()
{
    int voto;
    printf("Inserire un voto (fra 0 e 33): ");
    scanf("%d", &voto);
    if (voto < 18)
    {
        printf("Bocciato");
    }
    else
    {
        printf("Promosso");
        if (voto > 30)
        {
            printf(" con Lode");
        }
    }
}
```

Come sarebbe con
l'espressione
condizionale?

Blocco
innestato

18

Secondo piccolo problema: Ottimismo & Pessimismo

- Valutare il grado di ottimismo di una persona
- Richiedere l'inserimento di due valori
 - Se il secondo è minore del primo → pessimismo
 - Se il secondo è maggiore del primo → ottimismo
 - Se sono uguali → realismo

19

Ottimismo & Pessimismo

- Il problema è semplice e le specifiche sono già in pseudocodice...
- ...anche se sarebbe possibile raffinare ulteriormente:
 - Controlli d'errore
 - Che cosa succede se l'utente inserisce stringhe alfanumeriche anziché interi?
(questo è vero pessimismo ☹...)
- A voi la soluzione!

20

Assegnamento vs. Confronto

- **Assegnamento e confronto sono due operatori DIVERSI.** Errore pericoloso!

- Non causa errori di sintassi
- **Ogni espressione** che produce un valore può essere usata come **condizione in una struttura di controllo**
- Valori diversi da zero sono `true`, valori uguali a zero sono `false`

- Esempio usando `==`

```
if ( payCode == 4 )  
    printf( "Hai ottenuto un bonus!\n" );
```

- Controlla il `payCode`, se vale 4 allora viene concesso un bonus

21

Assegnamento vs. Confronto

- Per esempio, sostituendo `==` con `=`

```
if ( payCode = 4 )  
    printf( "Hai ottenuto un bonus!\n" );
```

- Assegna a `payCode` il valore 4
- L'assegnamento, come tutti gli statement, restituisce un valore, in particolare il valore assegnato; 4 è diverso da zero quindi l'espressione è `true` e il bonus è assegnato qualsiasi fosse il valore di `payCode`
- **è un errore LOGICO e non di SINTASSI!**

22

Selezione Multipla - switch

- **switch**

- Utile quando una variabile o espressione deve dar luogo a diverse azioni per i diversi valori assunti

- **Formato**

- Una serie di “etichette” **case** (caso) e una etichetta (un caso) **default** opzionale

```
switch ( value )
{
    case '1':
        Azioni;
        break;
    case '2':
        Azioni;
        break;
    default:
        Azioni;
        break;
}
```

- **break;** esce dallo statement

23

Menu

- All'interno di un'applicazione, dà all'utente la **possibilità di scegliere** quale azione compiere
- Nei programmi a console (interfaccia testuale) si stampano a video le varie possibilità poi si attende un input dall'utente
- A seconda di ciò che l'utente ha digitato, si esegue una particolare azione

24

Menu

1. Stampare **tutte le opzioni possibili** (compreso il comando di uscita) e un messaggio per far capire all'utente che cosa debba fare...

```
printf("1: Opzione 1\n");  
printf("2: Opzione 2\n");  
printf("0: Esci\n");  
printf("\nScegli un'opzione: ");
```

2. **Attendere la scelta** dell'utente

```
scanf("%d", &option);
```

25

Menu

3. **Selezionare** l'azione da compiere

```
switch (option)  
{  
    case 1:  
        printf("Opzione 1");  
        break;  
    case 2:  
        printf("Opzione 2");  
        break;  
    case 0:  
        printf("Uscita");  
        break;  
    default:  
        printf("Opzione errata");  
        break;  
}
```

26

Menu

4. Se non è stata scelta l'opzione di uscita, ***ricominciare da capo...***

- ***Serve un'istruzione di iterazione***

- Ad esempio ciclo **while** di ripetizione

- Il programmatore specifica una azione che deve essere ripetuta mentre (**while**) una certa espressione logica ***rimane true***
- Il loop **while** viene ripetuto finché la condizione diventa **false**

27

Somma dei primi n numeri

- Predisporre le **variabili** necessarie a contenere:
 - numero intero da raggiungere (da chiedere all'utente)
 - contatore: conta da 1 al numero suddetto
 - accumulatore della somma
- Chiedere all'utente di inserire un numero intero
- Leggere il numero intero
- Azzerare accumulatore e contatore
- Fintanto che il contatore è inferiore o uguale al numero inserito
 - Sommare il contatore all'accumulatore
 - Incrementare il contatore
- Stampare il risultato

28

Somma dei primi n numeri

- Ok, fatelo voi...
- ...ma un piccolo (molto piccolo) suggerimento non si nega a nessuno 😊...

```
while (counter <= n)
{
    ...
}
```

29

Ciclo do...while

- Altro costrutto sintattico di ripetizione
 - Il programmatore specifica una azione che deve essere **ripetuta mentre** (**while**) una certa espressione logica **rimane true**
 - A differenza del while semplice, **condizione in fondo**, quindi il corpo del ciclo viene eseguito almeno una volta
 - Il loop **do...while** viene ripetuto finché la condizione diventa **false**

30

Lettura Controllata

- Chiedere all'utente l'inserimento di dati da tastiera è un'operazione semplice ma va fatta con criterio
 - Che cosa succede se gli si chiede l'inserimento di un numero intero e lui inserisce qualcos'altro?
 - Come leggere sequenze di valori?

31

Lettura Controllata Controllo d'Errore

1. Richiedere l'inserimento di un **valore intero**
2. Attendere l'inserimento del valore da parte dell'utente
3. Il valore inserito è un valore intero?
 - Se non lo è, segnalare l'errore e chiedere all'utente se annullare l'operazione
 - In caso negativo riprendere dal punto 1
 - In caso positivo terminare la lettura

Note

- Che tipo di ciclo iterativo utilizzare?
...while? ...do...while?
- Come accorgersi che la lettura non è andata a buon fine?

32

Lettura Controllata

Controllo d'Errore

- Che cosa c'è dentro al ciclo?
 - **Lettura del valore da tastiera**
 - **Verifica correttezza** del valore letto
- Quante volte va eseguito il ciclo?
 - **Almeno una volta senza condizioni**
 - ...poi tutte le volte che è necessario
- Quindi, quale ciclo è più conveniente usare?
- Come accorgersi che la lettura **non è andata a buon fine**?

La funzione `scanf()` restituisce un intero che indica **quante sono le variabili lette con successo**

→ Non indica dove si sia verificato l'errore di lettura

33

Lettura Controllata - Pseudocodice

1. Dichiarare le variabili necessarie
 - **n**: valore letto
 - **success**: lettura effettuata con successo (o meno)
 - **cancel**: lettura annullata dall'utente
1. ----- Iniziare il ciclo
2. Stampare a video la richiesta di inserimento di un valore intero
3. Leggere il valore digitato dall'utente (inserire nella variabile **n**) e inserire il conteggio dei valori convertiti con successo nella variabile **success**
4. Se **success** vale 0 (nessun valore convertito) iniziare il trattamento dell'errore

34

Lettura Controllata - Pseudocodice

5. Trattamento dell'errore

1. Dichiarare una variabile (`op`) che possa contenere una risposta sì/no dell'utente
2. Chiedere all'utente se voglia annullare l'operazione
3. Leggere la risposta dell'utente (inserire nella variabile `op`)
4. Se `op` contiene una risposta affermativa mettere a `true` la variabile `cancel`, `false` altrimenti

6. Continuare il ciclo se nessun valore convertito (`success == 0`) e l'utente non ha richiesto la terminazione (`!cancel`)

35

Lettura Controllata

```
int n, success = 0, cancel = 0;
do
{
    printf("Inserisci un intero: ");
    success = scanf("%d", &n);
    if (success == 0)
    {
        char op;
        printf("Il valore inserito non è intero!");
        printf("\n");
        printf("Annullare l'operazione? (s/n)");
        while (getchar() != 10);    //Svuota il buffer
        scanf("%c", &op);
        getchar(); //Mangia solo il fine linea
        cancel = (op == 's' || op == 'S');
    }
}
while (success == 0 && !cancel);
```

36

Lettura controllata – i perché

- Perché sono necessari:
 - `while (getchar() != 10);`
 - `getchar(); //dopo scanf("%c",...);`
- Se la lettura non va a buon fine, `scanf()` lascia nel buffer i caratteri non consumati (e anche se va a buon fine...)
 - Per continuare a lavorare correttamente con il buffer di ingresso, **questi caratteri vanno eliminati**
 - In una linea inserita, l'ultimo carattere è sempre il **carattere 10 (LF – Line Feed)** generato dal tasto "invio"
 - `while` termina quando incontra l'ultimo carattere della linea (appunto il 10)

37

Lettura Controllata – i perché

- Se la lettura va a buon fine, il carattere LF (10) rimane nel buffer
- Ovviamente, anche nel caso di lettura di carattere ("`%c`"), LF rimane nel buffer
- Ulteriori informazioni quando si vedrà l'input/output in modo dettagliato ...

38

Ciclo for

```
for (inizializzazione; testDiContinuazione; ultimalstruzioneBlocco )  
    statement;
```

Equivalente a:

```
inizializzazione;  
while (testDiContinuazione)  
{  
    statement;  
    ultimalstruzioneBlocco;  
}
```

Tipicamente:

```
int counter;  
for (counter = 1; counter <= 10; counter++)  
{  
    doSomethingWithTheCounter;  
}
```

39

Tavola Pitagorica

- Dato un **fattore massimo**, scrivere a video la tavola pitagorica con fattori da 1 al valore inserito
- **Due cicli for innestati**: uno per le righe e uno per le colonne (ognuno col proprio contatore)
- Nel ciclo più interno si **stampa il risultato della moltiplicazione fra i due contatori**
- Possibili problemi di allineamento nella stampa: il risultato c'è ma è brutto...

Tavola Pitagorica – Pseudo e codice

1. Lettura fattore massimo (`maxFactor`)
2. Ciclo con indice `i` per righe da 1 a fattore massimo
 1. Ciclo con indice `j` per colonne da 1 a fattore massimo
 1. Stampa `i * j`

...

```
int maxFactor, i, j;  
...lettura maxFactor...  
for (i = 1; i <= maxFactor; i++)  
    for (j = 1; j <= maxFactor; j++)  
        printf(" %d", i * j);
```

...

41

Tavola Pitagorica ...e l'allineamento?

- Calcolare l'occupazione massima in termini di cifre degli interi da stampare e adattare la stampa di volta in volta
 - Qual è l'intero più grande (quello che occupa più spazio)?
 - `maxFactor * maxFactor`
 - Di quante cifre è composto?
 - `(int)log10(maxFactor * maxFactor)`
- Un'idea potrebbe essere di inserire davanti alla cifra da stampare ***tanti spazi bianchi*** quanti sono necessari affinché tutti i valori stampati occupino lo stesso spazio
 - Cifre di cui è composto il valore da stampare:
 - `(int)log10(i * j)`
 - Spazi bianchi necessari:
 - `(int)log10(maxFactor * maxFactor) - (int)log10(i * j)`

42