

Programmare in C++

A.S. 2015/2016

Alessandro Saltini

Liceo Scientifico Statale "A. Tassoni"

- ▶ L'informatica **non** è
 - ▶ saper usare un computer
 - ▶ saper costruire/riparare un computer
 - ▶ usare programmi scritti da altri
- ▶ L'informatica **è**
 - ▶ una branca della matematica
 - ▶ lo studio dell'**informazione**
 - ▶ lo studio degli **algoritmi**
 - ▶ lo studio dei **linguaggi di programmazione**

- ▶ L'informazione si misura in **bit** (binary digit)
- ▶ 1 bit è la quantità di informazione necessaria a determinare una quantità che può essere **0** o **1**
- ▶ Il **byte** è un multiplo del bit: $1 \text{ B} = 8 \text{ bit}$
- ▶ Due scale di multipli del byte:
 - ▶ decimale: kB (10^3), MB (10^6), GB (10^9), TB (10^{12}), ...
 - ▶ **binaria**: KiB (2^{10}), MiB (2^{20}), GiB (2^{30}), TiB (2^{40}), ...

- ▶ L'algebra Booleana è l'algebra dei bit
- ▶ È un **modello** della logica classica: 1 = vero, 0 = falso
- ▶ Insieme di base $B = \{ 0, 1 \}$
- ▶ Tre operazioni fondamentali:
 - ▶ **not** (non): $\neg : B \rightarrow B$
 - ▶ **and** (et): $\wedge : B^2 \rightarrow B$
 - ▶ **or** (vel): $\vee : B^2 \rightarrow B$

► not (non): \neg

► $\neg 1 = 0$

► $\neg 0 = 1$

► and (et): \wedge

► $1 \wedge 1 = 1$

► $1 \wedge 0 = 0$

► $0 \wedge 1 = 0$

► $0 \wedge 0 = 0$

► or (vel): \vee

► $1 \vee 1 = 1$

► $1 \vee 0 = 1$

► $0 \vee 1 = 1$

► $0 \vee 0 = 0$

- ▶ Combinando queste tre operazioni si possono ottenere tutte le altre operazioni possibili
- ▶ In realtà basta **una** sola operazione, meno intuitiva:
 - ▶ nand (\uparrow)
 - ▶ nor (\downarrow)
- ▶ Esistono circuiti **elettrici** che realizzano materialmente queste operazioni logiche
 - ▶ segnale "alto" = 1
 - ▶ segnale "basso" = 0
- ▶ Sono l'elemento di base dei computer

- ▶ Un numero in rappresentazione **decimale** è espresso come combinazione di potenze di 10

$$1064 = 1 \cdot 10^3 + 0 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0$$

- ▶ I coefficienti sono compresi tra 0 e 9 (**minori** di 10)
- ▶ Il massimo numero con n cifre decimali è $10^n - 1$
- ▶ I numeri esistono indipendentemente dalla loro rappresentazione, è solo un modo di scriverli

- ▶ La rappresentazione **binaria** utilizza le potenze di 2

$$10110 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

- ▶ I coefficienti sono soltanto 0 e 1 (**minori** di 2)

- ▶ Il massimo numero con n cifre binarie è $2^n - 1$

- ▶ Ogni cifra è rappresentabile da un **bit**

- ▶ n cifre $\Rightarrow n$ bit

- ▶ I computer memorizzano i numeri in binario
- ▶ Le operazioni tra essi vengono svolte da appositi circuiti, basati sulle operazioni Booleane
- ▶ Ogni operazione richiede un certo tempo
- ▶ Limiti di memoria/tempo impediscono di operare con numeri arbitrariamente grandi

- ▶ I numeri negativi devono memorizzare anche il segno
- ▶ Costo di 1 bit aggiuntivo
 - ▶ $s = 0 \Rightarrow +$
 - ▶ $s = 1 \Rightarrow -$
- ▶ Spesso si ricorre a rappresentazioni alternative
 - ▶ rimozione di ambiguità tra $+0$ e -0
 - ▶ facilità di calcolo
 - ▶ occupano comunque 1 bit in più

- ▶ La parte frazionaria è problematica da rappresentare

$$1.1011 = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}$$

- ▶ Non tutti i numeri con rappresentazione decimale finita hanno rappresentazione binaria finita
- ▶ Non possiamo memorizzare infinite cifre
 - ▶ Impossibile rappresentare i numeri irrazionali
 - ▶ Non tutti i numeri razionali sono rappresentabili

- ▶ Richiamiamo la notazione scientifica

$$1064.15 = 1.06415 \cdot 10^3$$

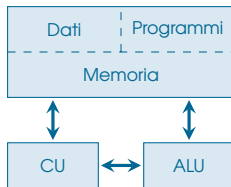
- ▶ Generalizzabile in binario come

$$110.1011 = 1.101011 \cdot 2^2$$

- ▶ La **prima** cifra della rappresentazione scientifica binaria è **sempre 1**, non serve memorizzarla

$$110.1011 = 1.101011 \cdot 2^2$$

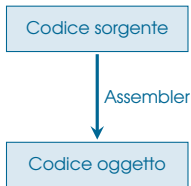
- ▶ La parte dopo la virgola è detta **mantissa** o **significando**, è un numero intero
- ▶ L'**esponente** di 2 è un numero intero
- ▶ Un numero frazionario viene rappresentato come coppia di numeri interi
 - ▶ bit del significando \Rightarrow precisione
 - ▶ bit dell'esponente \Rightarrow range



- ▶ Memoria unica per dati e programmi
- ▶ CU (Control Unit): assegna e gestisce risorse
- ▶ ALU (Arithmetic Logic Unit): compie operazioni
- ▶ ALU + CU = CPU (Central Processing Unit)

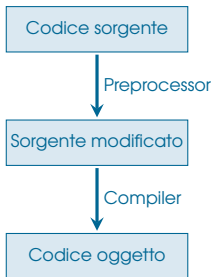
- ▶ La CPU esegue istruzioni **semplici**
- ▶ Istruzione tipo:
 - ▶ che operazione fare
 - ▶ indirizzi di memoria degli operandi
 - ▶ indirizzi di memoria dei risultati
- ▶ Le istruzioni sono codificate come numeri binari
- ▶ Le istruzioni eseguibili dipendono dal tipo di CPU

- ▶ I linguaggi **assembly** hanno una corrispondenza 1-ad-1 con le istruzioni della macchina
- ▶ Le istruzioni vengono “tradotte” da numeri binari a parole comprensibili ad un essere umano
- ▶ Un programma in assembly è una lista di istruzioni che la CPU eseguirà nell'ordine in cui appaiono
- ▶ CPU diverse hanno linguaggi assembly diversi

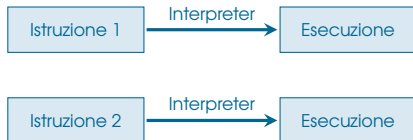


- ▶ Il **codice sorgente** è un file contenente testo
- ▶ Il **codice oggetto** è un file contenente istruzioni binarie
- ▶ Un programma detto **assembler** converte il codice sorgente in codice oggetto, eseguibile dalla CPU

- ▶ Un **linguaggio** di programmazione è un linguaggio **formale** con il quale scrivere istruzioni
 - ▶ distinzione tra istruzioni valide/invalidi
 - ▶ le istruzioni controllano la CPU
- ▶ Distinzione di **livello**
 - ▶ basso livello: **1-ad-1** con istruzioni della CPU
 - ▶ alto livello: linguaggi **astratti**
- ▶ Un **paradigma di programmazione** è uno stile secondo il quale viene scritto il codice sorgente
- ▶ Ciascun linguaggio accetta uno o più paradigmi



- ▶ Il codice sorgente viene scritto per intero
- ▶ Il **preprocessor** modifica al sorgente (facoltativo)
- ▶ Il **compiler** crea un file oggetto dal sorgente



- ▶ Ciascuna istruzione viene compilata singolarmente
- ▶ Esecuzione di programmi incompleti
- ▶ Molto più lenti dei linguaggi compilati

- ▶ Programmazione imperativa
 - ▶ lista di **comandi** eseguiti in un certo ordine
- ▶ Programmazione dichiarativa
 - ▶ lista di **relazioni** tra enti
- ▶ Programmazione ad oggetti
 - ▶ lista di **oggetti** e di **interazioni** tra essi

- ▶ Scrittura di un **algoritmo** per risolvere un problema
- ▶ Esempio: cambiare la batteria di un telecomando
 - ▶ aprire il vano batterie
 - ▶ rimuovere la vecchia batteria
 - ▶ gettare la vecchia batteria
 - ▶ inserire la nuova batteria
 - ▶ chiudere il vano batterie
- ▶ L'ordine delle istruzioni è determinante

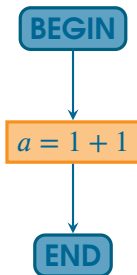
- ▶ Per rappresentare un algoritmo si utilizzano i flowcharts o diagrammi di flusso
- ▶ Un flowchart è costituito da
 - ▶ celle contenenti istruzioni
 - ▶ frecce che guidano il flusso di controllo
- ▶ I flowcharts sono un linguaggio di programmazione
- ▶ Creare flowcharts aiuta nella stesura di un algoritmo



- ▶ Istruzioni di inizio e fine diagramma
- ▶ Uno ed un solo BEGIN per diagramma
- ▶ A volte ammessi più END in un singolo diagramma



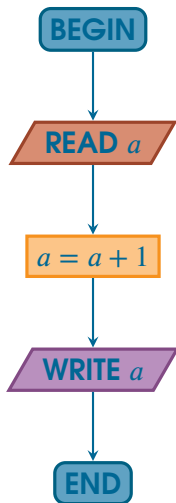
- ▶ Istruzione di processo
- ▶ Viene eseguita l'operazione indicata nella casella

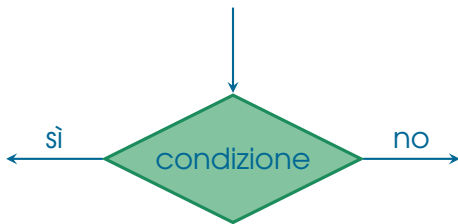


- ▶ Il programma calcola $1+1$
- ▶ Il risultato viene messo nella **variabile** a

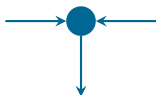


- ▶ Istruzioni di lettura/scrittura
- ▶ Lettura: un valore inserito dall'**utente** viene memorizzato nella variabile x
- ▶ Scrittura: il valore corrente della variabile x viene comunicato all'**utente**

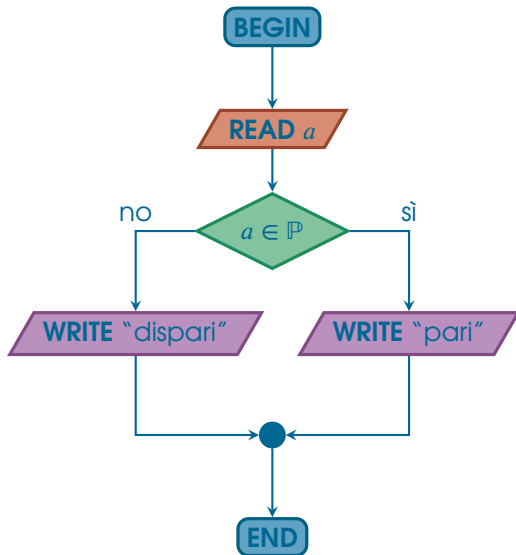




- ▶ Istruzione condizionale
- ▶ Il flusso del grafico cambia direzione a seconda che la condizione sia vera oppure falsa



- ▶ Connettore
- ▶ Permette di rincongiungere due rami separati

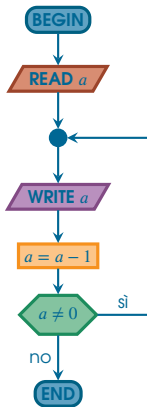


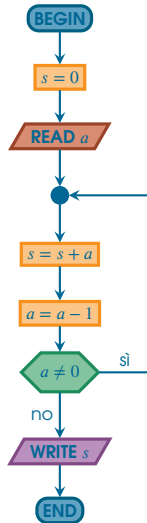
- ▶ Le regole elencate danno origine alla programmazione imperativa in senso lato
- ▶ Le frecce possono “risalire” il diagramma
 - ▶ istruzioni **goto**: ritorno ad un punto precedente
- ▶ Questo rende più difficile:
 - ▶ studio formale del programma
 - ▶ apportare modifiche al codice
 - ▶ comprensione del programma da parte di terzi

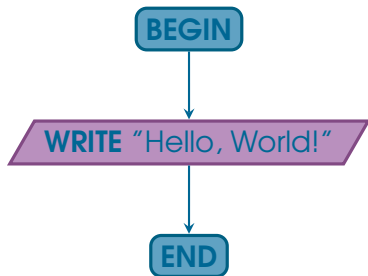
- ▶ La programmazione **strutturata** vieta i goto
- ▶ Le frecce non possono risalire il diagramma
- ▶ Viene aggiunto un nuovo simbolo
- ▶ Teorema di Böhm-Jacopini
 - ▶ **equivalenza** tra imperativa e strutturata



- ▶ Istruzione di loop
- ▶ Concettualmente identica all'istruzione condizionale, ma uno (ed uno solo) dei due flussi può risalire
- ▶ Permette di ripetere una serie di istruzioni







```
1 | #include <iostream>
2 |
3 | int main() {
4 |     std::cout << "Hello, World!" << std::endl;
5 |     return 0;
6 | }
7 |
```

- ▶ Il codice sorgente viene scritto in un file di testo
 - ▶ estensioni standard `*.cpp` o `*.cc` o `*.C`
- ▶ La compilazione può avvenire in due modi
 - ▶ tramite IDE (ambiente di sviluppo)
 - ▶ `g++ -std=c++14 source.cpp -o target`
- ▶ Per progetti più ampi conviene creare un **Makefile**

```
1 | #include <iostream>
2 |
3 | int main() {
4 |     std::cout << "Hello, World!" << std::endl;
5 |     return 0;
6 | }
7 |
```

- ▶ **#** introduce direttive del preprocessore
- ▶ **int** `main()` è la parte principale del programma
- ▶ `{ ... }` raggruppa più istruzioni in un singolo blocco
- ▶ ogni istruzione termina con un `;`
- ▶ ogni programma deve terminare con **return** `0;`
- ▶ l'ultima riga di un file deve sempre essere vuota

```
1 | #include <iostream>
2 |
3 | int main() {
4 |     std::cout << "Hello, World!" << std::endl;
5 |     return 0;
6 | }
7 |
```

- ▶ **#include** <...> inserisce un **header** nel programma
 - ▶ funzioni di libreria (C++ Standard Library)
 - ▶ tutte nel **namespace** `std`
- ▶ `<iostream>` contiene:
 - ▶ `std::cout` necessaria per scrivere a schermo
 - ▶ `std::endl` inserisce un'interruzione di riga


```
1 | #include <iostream>
2 |
3 | int main() {
4 |     std::cout << "Hello, World!" << std::endl;
5 |     return 0;
6 | }
7 |
```

- ▶ C++ gestisce l'output tramite **stream** (flussi)
 - ▶ `std::cout` è il flusso di output standard
 - ▶ `<<` è l'operatore di **inserimento**
 - ▶ `std::endl` è un **manipolatore**
- ▶ `"Hello, World!"` è una **stringa**
 - ▶ delimitate da **virgolette**

```
1 //Questo è il mio primo programma in C++
2 #include <iostream>
3
4 int main() {
5     std::cout << "Hello, World!" << std::endl;
6     return 0;
7 }
8
```

- ▶ le righe introdotte da `//` sono **commenti**
- ▶ commentare è una buona abitudine
 - ▶ aiuta a ricordare cosa fa il programma
 - ▶ aiuta altri a capire cosa fa il programma
- ▶ altro modo di creare commenti: `/* ... */`

```
1 //Questo è il mio primo programma in C++
2 #include <iostream>
3 using namespace std;
4 int main() {
5     cout << "Hello, World!" << endl;
6     return 0;
7 }
8
```

- ▶ Importa **tutte** le funzioni del namespace std
- ▶ Comodo per non scrivere `std::` ogni volta
- ▶ Cattiva abitudine, molto rischioso

```
1 //Questo è il mio primo programma in C++
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5 int main() {
6     cout << "Hello, World!" << endl;
7     return 0;
8 }
9
```

- ▶ Importa soltanto le funzioni che richiediamo
- ▶ Più sicuro che importare l'intero namespace

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main() {
5      int a;
6      a = 5;
7      cout << a << endl;
8      return 0;
9  }
10
```



- ▶ `int a;` dichiara la variabile `a` di tipo `int`
- ▶ `a = 5;` assegna il valore 5 alla variabile `a`
 - ▶ la prima assegnazione è detta **inizializzazione**
 - ▶ **mai** utilizzare una variabile non inizializzata

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int a;
5 int main() {
6     a = 5;
7     cout << a << endl;
8     return 0;
9 }
10
```



- ▶ Dichiarazione all'esterno di `main()`
- ▶ Variabile **globale**
- ▶ Ammesso, ma fortemente sconsigliato

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a = 5;
6     cout << a << endl;
7     return 0;
8 }
9
```



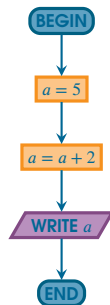
- ▶ Inizializzazione in sede di dichiarazione
- ▶ Buona abitudine

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     a = 5 + 2;
7     cout << a << endl;
8     return 0;
9 }
10
```



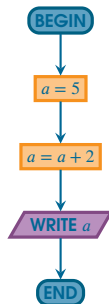
► È possibile compiere operazioni


```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a = 5;
6     a = a + 2;
7     cout << a << endl;
8     return 0;
9 }
10
```



- ▶ Utilizzo del valore di una variabile in un'operazione:
 - ▶ leggo il valore di a
 - ▶ sommo 2 al valore letto
 - ▶ metto il risultato in a

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a = 5;
6     a += 2;
7     cout << a << endl;
8     return 0;
9 }
10
```



► Forma più compatta della scrittura precedente

► Operazioni:

+	somma	$5 + 3 = 8$
-	sottrazione	$5 - 3 = 2$
*	moltiplicazione	$5 * 3 = 15$
/	divisione	$5 / 3 = 1$
%	resto	$5 \% 3 = 2$

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     int b;
7     a = 5;
8     b = 3;
9     cout << a << endl;
10    cout << b << endl;
11    return 0;
12 }
13
```



► Posso dichiarare più variabili

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a, b;
6     a = 5;
7     b = 3;
8     cout << a << endl;
9     cout << b << endl;
10    return 0;
11 }
12
```



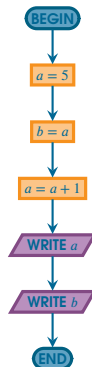
- ▶ Variabili dello stesso tipo possono essere dichiarate in un'unica istruzione, separate da virgole

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a = 5, b = 3;
6     cout << a << endl;
7     cout << b << endl;
8     return 0;
9 }
10
```



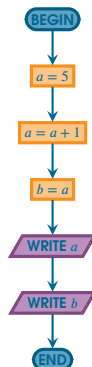
► Inizializzazione in sede di dichiarazione

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main() {
5      int a = 5, b;
6      b = a++;
7      cout << a << endl;
8      cout << b << endl;
9      return 0;
10 }
11
```



- ▶ `a++` operatore di post-incremento
- ▶ `a--` operatore di post-decremento

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main() {
5      int a = 5, b;
6      b = ++a;
7      cout << a << endl;
8      cout << b << endl;
9      return 0;
10 }
11
```



- ▶ ++a operatore di pre-incremento
- ▶ --a operatore di pre-decremento

- ▶ Tipi di variabile:
 - ▶ numeri interi
 - ▶ **short**
 - ▶ **int**
 - ▶ **long**
 - ▶ **long long**
 - ▶ numeri razionali
 - ▶ **float**
 - ▶ **double**
 - ▶ **long double**
 - ▶ booleano (0/1)
 - ▶ **bool**
 - ▶ carattere
 - ▶ **char**

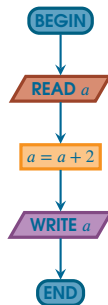
```
1 #include <iostream>
2 #include <limits>
3 using std::cout;
4 using std::endl;
5 int main() {
6     cout << std::numeric_limits<int>::max() << endl;
7     cout << std::numeric_limits<int>::min() << endl;
8     return 0;
9 }
10
```

- ▶ La dimensione dei tipi non è standard
- ▶ Ogni compilatore può avere dimensioni diverse
- ▶ È bene verificare i limiti del proprio compilatore

```
1 | int main() {  
2 |     const int a = 5;  
3 |     a = 4;  
4 |     return 0;  
5 | }  
6 |
```

- ▶ La parola **const** impedisce di cambiare una variabile
- ▶ Il valore deve essere impostato nella dichiarazione
- ▶ Utile per evitare di cambiare accidentalmente quantità che devono restare fisse

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     std::cin >> a;
7     a += 2;
8     cout << a << endl;
9     return 0;
10 }
11
```



- ▶ `std::cin` è il flusso di input standard
- ▶ `>>` è l'operatore di estrazione

```
1 #include <iostream>
2 int main() {
3     std::cerr << "Errore!" << std::endl;
4     return 1;
5 }
6
```

- ▶ `std::cerr` è il flusso su cui comunicare gli errori
- ▶ Viene gestito diversamente dal sistema operativo

```
1 #include <fstream>
2 int main() {
3     std::ofstream fout;
4     fout.open("prova.txt");
5     fout << "Hello, World!" << std::endl;
6     fout.close();
7     return 0;
8 }
9
```

- ▶ `<fstream>` permette di fare input/output su file
- ▶ `std::ofstream` è un tipo di variabile
 - ▶ `fout` è il nome della variabile

```
1 #include <fstream>
2 int main() {
3     std::ofstream fout;
4     fout.open("prova.txt");
5     fout << "Hello, World!" << std::endl;
6     fout.close();
7     return 0;
8 }
9
```

- ▶ `fout.open("prova.txt")` apre il file `prova.txt`
- ▶ La scrittura avviene come per `cout`
- ▶ `fout.close()` chiude il file

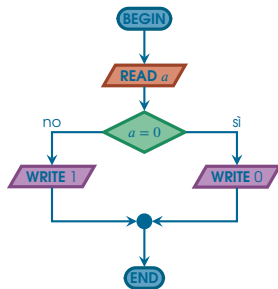
```
1  #include <fstream>
2  int main() {
3      std::ofstream fout("prova.txt");
4      fout << "Hello, World!" << std::endl;
5      fout.close();
6      return 0;
7  }
8
```

- ▶ L'apertura del file può essere fatta nella dichiarazione


```
1 #include <iostream>
2 #include <fstream>
3 int main() {
4     int a;
5     std::ifstream fin("dati.txt");
6     fin >> a;
7     fin.close();
8     std::cout << a << std::endl;
9     return 0;
10 }
11
```

- ▶ `std::ifstream` per i file di input
- ▶ L'operatore `>>` estrae il primo dato nel file

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     std::cin >> a;
7     if(a == 0) {
8         cout << 0 << endl;
9     } else {
10        cout << 1 << endl;
11    }
12    return 0;
13 }
14
```



- ▶ `if(condizione) { ... } else { ... }`
 - ▶ il primo blocco viene eseguito se la condizione è vera
 - ▶ il secondo blocco viene eseguito se è falsa
- ▶ La condizione deve avere un valore di tipo `bool`
 - ▶ spesso risulta da operatori di **confronto**
 - ▶ può contenere espressioni composte
- ▶ La direttiva `else` può essere omessa

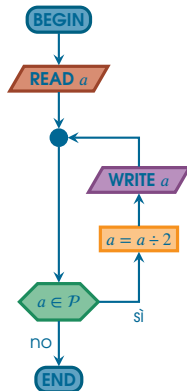
► Operatori di confronto:

==	uguale
!=	diverso
>	maggiore
>=	maggiore o uguale
<	minore
<=	minore o uguale

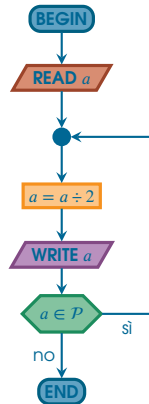
► Operatori logici:

!	not
&&	and
	or

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     std::cin >> a;
7     while(a % 2 == 0) {
8         a /= 2;
9         cout << a << endl;
10    }
11    return 0;
12 }
13
```

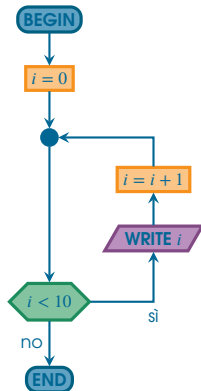


```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     std::cin >> a;
7     do {
8         a /= 2;
9         cout << a << endl;
10    } while(a % 2 == 0);
11    return 0;
12 }
13
```

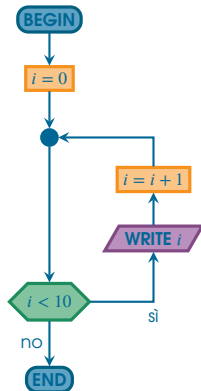


- ▶ **while**(condizione) { ... }
 - ▶ ripete il blocco finché la condizione è vera
 - ▶ se la condizione è inizialmente falsa, non entra
- ▶ **do** { ... } **while**(condizione);
 - ▶ ripete il blocco finché la condizione è vera
 - ▶ il blocco viene eseguito **almeno** una volta
- ▶ L'**iterazione** è alla base della programmazione
 - ▶ sfruttare il computer per fare operazioni ripetitive

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int i = 0;
6     while(i < 10){
7         cout << i << endl;
8         i++;
9     }
10    return 0;
11 }
12
```




```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     for(int i = 0; i < 10; i++){
6         cout << i << endl;
7     }
8     return 0;
9 }
10
```



- ▶ **for** (iniziale; condizione; incremento) { ... }
 - ▶ esegue il comando iniziale
 - ▶ ripete il blocco finché la condizione è vera
 - ▶ al termine di ogni iterazione esegue l'incremento

- ▶ Utile nel caso sia noto a priori il numero di ripetizioni che un ciclo deve compiere

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     for(int i = 0; i < 10; i++){
6         i *= 2;
7         cout << i << endl;
8     }
9     return 0;
10 }
11
```

- ▶ Cattiva abitudine: cambiare l'indice dentro al ciclo
- ▶ Cambiare la struttura del ciclo o usare un while

```
1 #include <iostream>
2 int main() {
3     signed int a = -50;
4     unsigned int b = -50;
5     std::cout << a << std::endl;
6     std::cout << b << std::endl;
7     return 0;
8 }
9
```

- ▶ **unsigned** è un tipo di intero senza segno
 - ▶ range doppio, bit del segno usato per le cifre
 - ▶ comportamento inatteso con numeri negativi
- ▶ **signed** è generalmente sottointeso

```
1 #include <iostream>
2 typedef unsigned int uint;
3 int main() {
4     uint a = 6;
5     std::cout << a << std::endl;
6     return 0;
7 }
8
```

- ▶ **typedef** definisce un nuovo tipo
 - ▶ abbrevia la scrittura
 - ▶ modifica flessibile su programmi lunghi
- ▶ Alcuni compilatori definiscono già `uint`, ma è consigliabile ridefinirlo con un `typedef` per portabilità

```
1 #include <iostream>
2 int main() {
3     char a = 'A';
4     std::cout << a << std::endl;
5     std::cout << static_cast<int>(a) << std::endl;
6     std::cout << dynamic_cast<int>(a) << std::endl;
7     return 0;
8 }
9
```

- Conversione di dati da un tipo ad un altro
 - **static_cast**: più sicuro, più veloce
 - **dynamic_cast**: più flessibile
 - differenze irrisorie sui tipi standard

```
1 #include <iostream>
2 int main() {
3     char a = 'A';
4     std::cout << a << std::endl;
5     std::cout << (int) a << std::endl;
6     return 0;
7 }
8
```

- ▶ Legacy C-style cast: `(int) a`
 - ▶ sconsigliato, comportamento inaffidabile

- ▶ Matematica: $f : A \rightarrow B$
 - ▶ A dominio
 - ▶ B codominio
 - ▶ associa ciascun elemento di A ad un elemento di B
- ▶ Funzioni a più input: $f : A_1 \times A_2 \rightarrow B$
 - ▶ coppia di valori da A_1 e A_2 come input
 - ▶ più in generale, n-ple di valori
- ▶ Informatica: **tipi** prendono il posto degli insiemi

- ▶ Funzioni **pure**
 - ▶ non modificano i valori di input
 - ▶ il valore di output dipende **solo** dai valori di input
 - ▶ non producono lettura/scrittura a schermo
- ▶ Funzioni **impure**
 - ▶ possono modificare gli input
 - ▶ possono utilizzare valori casuali, date, ore
 - ▶ possono leggere o scrivere a schermo
- ▶ La sintassi del C++ non le distingue

```
1 #include <iostream>
2 int twice(int x) {
3     return 2*x;
4 }
5 int main() {
6     int a = 3;
7     std::cout << twice(a) << std::endl;
8     std::cout << a << std::endl;
9     return 0;
10 }
11
```

► $\text{twice} : \text{int} \rightarrow \text{int} :: x \mapsto 2x$

► $\text{main} : \text{void} \rightarrow \text{int}$

```
2 | int twice(int x) {  
3 |     return 2*x;  
4 | }
```

► Definizione della funzione

- il primo **int** è il tipo di output
- segue il nome della funzione
- l'**int** tra parentesi è il tipo di input
- x indica il nome della variabile
- il blocco { ... } è l'**implementazione**

```
7 | std::cout << twice(a) << std::endl;
```

► Chiamata alla funzione

```
1  #include <iostream>
2  int pow(int x, int y) {
3      int z = 1;
4      for(int i = 0; i < y; i++) z *= x;
5      return z;
6  }
7  int main() {
8      std::cout << pow(2,3) << std::endl;
9      return 0;
10 }
11
```

► $\text{pow} : \text{int} \times \text{int} \rightarrow \text{int} :: (x, y) \mapsto x^y$

► Funzione a due input

```
1 #include <iostream>
2 int two() { return 2; }
3 int main() {
4     std::cout << two() << std::endl;
5     return 0;
6 }
7
```

▶ `two : void → int :: 2`

▶ Funzione senza input

▶ Chiamata con `two()`

```
1 | #include <iostream>
2 | int twice(int);
3 | int main() {
4 |     int a = 3;
5 |     std::cout << twice(a) << std::endl;
6 |     std::cout << a << std::endl;
7 |     return 0;
8 | }
9 | int twice(int x) { return 2*x; }
10 |
```

► `int twice(int);` è la **dichiarazione**

► `int twice(int x) { return 2*x; }` è la **definizione**

- ▶ La dichiarazione dice che esiste una funzione
 - ▶ deve precedere il primo utilizzo della funzione
 - ▶ buona occasione per commentare
- ▶ La definizione implementa la funzione
 - ▶ senza di dichiarazione deve precedere il primo utilizzo
- ▶ Separarle è una buona abitudine

```
1  #include <iostream>
2  int fibonacci(int);
3  int main() {
4      std::cout << fibonacci(8) << std::endl;
5      return 0;
6  }
7  int fibonacci(int n) {
8      int f;
9      if (n == 0) f = 0;
10     else if (n == 1) f = 1;
11         else f = fibonacci(n-1) + fibonacci(n-2);
12     return f;
13 }
14
```

► Funzione **ricorsiva**: chiama sé stessa


```
1  #include <iostream>
2  void swap(int, int)
3  int main() {
4      int a = 1, b = 0;
5      std::cout << a << " " << b << std::endl;
6      swap(a, b);
7      std::cout << a << " " << b << std::endl;
8      return 0;
9  }
10 void swap(int x, int y) {
11     int z = x;
12     x = y;
13     y = z;
14     return;
15 }
16
```

```
1  #include <iostream>
2  void swap(int&, int&)
3  int main() {
4      int a = 1, b = 0;
5      std::cout << a << " " << b << std::endl;
6      swap(a,b);
7      std::cout << a << " " << b << std::endl;
8      return 0;
9  }
10 void swap(int& x, int& y) {
11     int z = x;
12     x = y;
13     y = z;
14     return;
15 }
16
```

```
1 #include <iostream>
2 int main() {
3     signed char a = 26;
4     a += 5;
5     std::cout << static_cast<int>(a) << std::endl;
6     a -= 42;
7     std::cout << static_cast<int>(a) << std::endl;
8     return 0;
9 }
10
```

- ▶ **signed/unsigned char** come intero
 - ▶ minor utilizzo di memoria
 - ▶ maggior tempo di calcolo (su macchine moderne)
 - ▶ no input diretto (serve variabile di appoggio)