

# 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

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

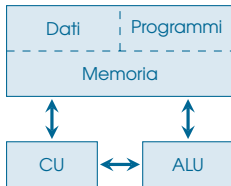
- ▶ Generalizzabile in binario come

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

- ▶ 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

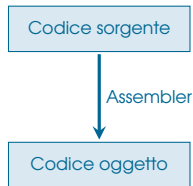


- ▶ Memoria unica per dati e programmi
- ▶ CU (Control Unit): assegna e gestisce risorse
- ▶ ALU (Arithmetic Logic Unit): compie operazioni
- ▶ ALU + CU = CPU (Central Processing Unit)

- ▶ 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

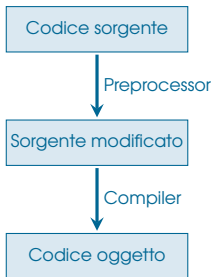
- ▶ I linguaggi **assembly** hanno una corrispondenza 1-ad-1 con le istruzioni della macchina
- ▶ Le istruzioni vengono “tradotte” da numeri binari a parole comprensibili ad un essere umano
- ▶ Un programma in assembly è una lista di istruzioni che la CPU eseguirà nell’ordine in cui appaiono
- ▶ CPU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



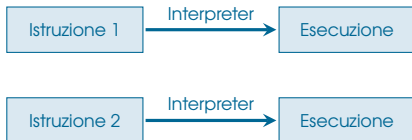


- ▶ Il **codice sorgente** è un file contenente testo
- ▶ Il **codice oggetto** è un file contenente istruzioni binarie
- ▶ Un programma detto **assembler** converte il codice sorgente in codice oggetto, eseguibile dalla CPU

- ▶ 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

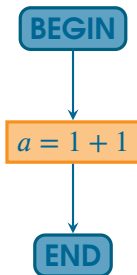


- ▶ Istruzioni di inizio e fine diagramma
- ▶ Uno ed un solo **BEGIN** per diagramma
- ▶ A volte ammessi più **END** in un singolo diagramma





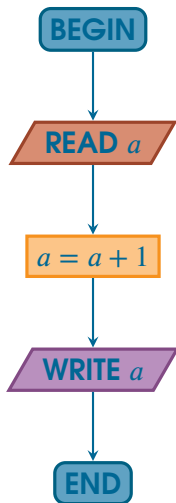
- ▶ Istruzione di processo
- ▶ Viene eseguita l'operazione indicata nella casella

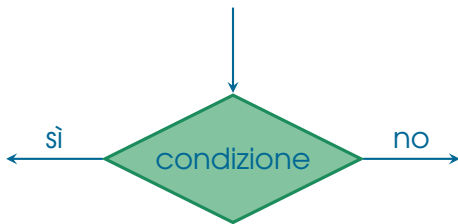


- ▶ Il programma calcola  $1+1$
- ▶ Il risultato viene messo nella **variabile**  $a$

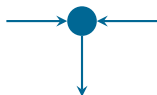


- ▶ Istruzioni di lettura/scrittura
- ▶ Lettura: un valore inserito dall'**utente** viene memorizzato nella variabile  $x$
- ▶ Scrittura: il valore corrente della variabile  $x$  viene comunicato all'**utente**

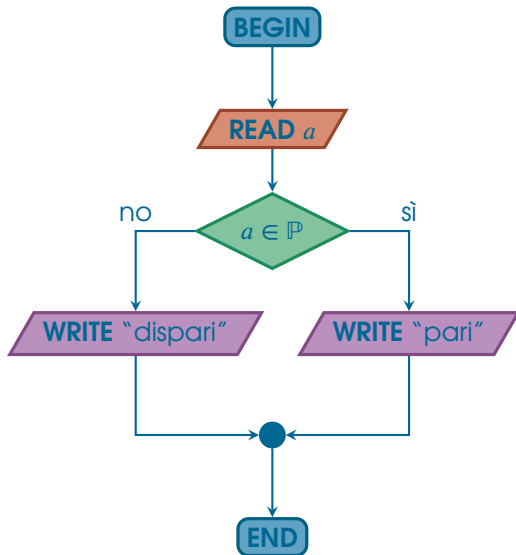




- ▶ Istruzione condizionale
- ▶ Il flusso del grafico cambia direzione a seconda che la condizione sia vera oppure falsa



- ▶ Connettore
- ▶ Permette di rincongiungere due rami separati



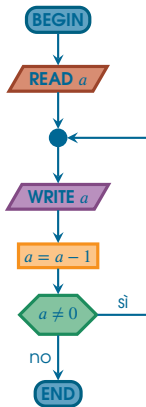
- ▶ Le regole elencate danno origine alla programmazione imperativa in senso lato
- ▶ Le frecce possono “risalire” il diagramma
  - ▶ istruzioni **goto**: ritorno ad un punto precedente
- ▶ Questo rende più difficile:
  - ▶ studio formale del programma
  - ▶ apportare modifiche al codice
  - ▶ comprensione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

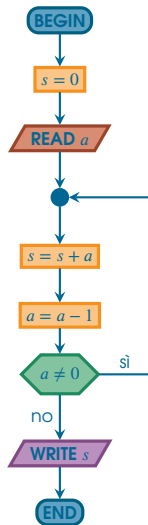


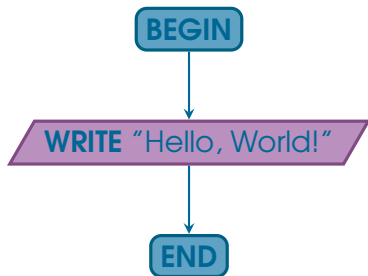
- ▶ La programmazione **strutturata** vieta i goto
- ▶ Le frecce non possono risalire il diagramma
- ▶ Viene aggiunto un nuovo simbolo
- ▶ Teorema di Böhm-Jacopini
  - ▶ **equivalenza** tra imperativa e strutturata



- ▶ 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(5);
6      cout << a << endl;
7      return 0;
8  }
9  
```



► Sintassi alternativa: **costruttore**

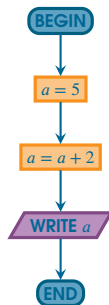


```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a;
6     a = 5 + 2;
7     cout << a << endl;
8     return 0;
9 }
10
```



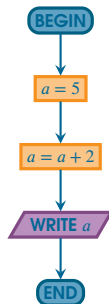
► È possibile compiere operazioni

```
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
```



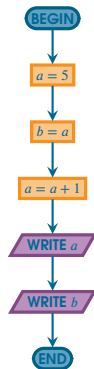
- ▶ Variabili dello stesso tipo possono essere dichiarate in un'unica istruzione, separate da virgole

```
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

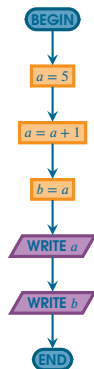
```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int a = 5, b;
6     b = a++;
7     cout << a << endl;
8     cout << b << endl;
9     return 0;
10 }
11
```



- ▶ `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`

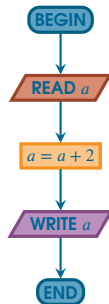
```
1 #include <iostream>
2 #include <limits>
3 using std::cout;
4 using std::endl;
5 int main() {
6     cout << std::numeric_limits<int>::max() << endl;
7     cout << std::numeric_limits<int>::min() << endl;
8     return 0;
9 }
10
```

- ▶ 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



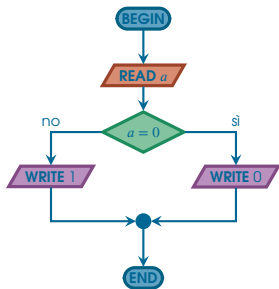
```
1 #include <fstream>
2 int main() {
3     std::ofstream fout("prova.txt");
4     fout << "Hello, World!" << std::endl;
5     fout.close();
6     return 0;
7 }
8
```

- 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

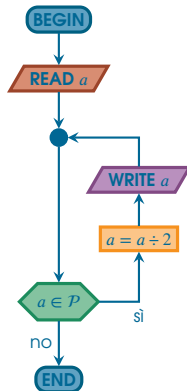
► Operatori di confronto:

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

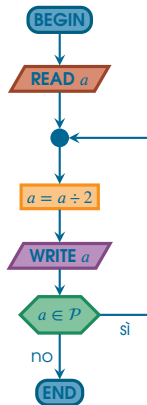
► Operatori logici:

! 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
```



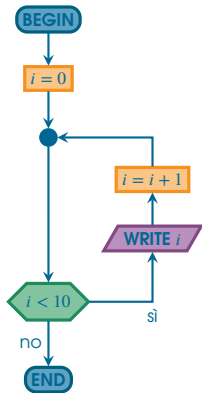
```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main() {
5      int a;
6      std::cin >> a;
7      do {
8          a /= 2;
9          cout << a << endl;
10     } while(a % 2 == 0);
11     return 0;
12 }
13
```



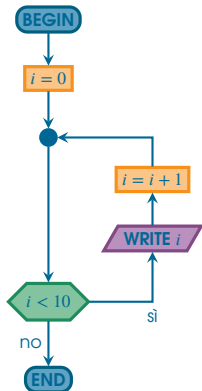
- ▶ **while**(condizione) { ... }
  - ▶ ripete il blocco finché la condizione è vera
  - ▶ se la condizione è inizialmente falsa, non entra
- ▶ **do** { ... } **while**(condizione);
  - ▶ ripete il blocco finché la condizione è vera
  - ▶ il blocco viene eseguito **almeno** una volta
- ▶ L'**iterazione** 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     return 0;
7 }
8
```

- Conversione di dati da un tipo ad un altro
  - **static\_cast**: più sicuro, più veloce
  - esistono altre funzioni di cast

```
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 o prototipo
- ▶ `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 hello(int);
3  int main() {
4      hello(5);
5      return 0;
6  }
7  void hello(int x) {
8      for(int i = 0; i < x; i++) {
9          std::cout << "Hello, World!" << std::endl;
10     }
11     return;
12 }
13
```

► Procedura: funzione senza tipo di output

- ▶ Passaggio di parametri **by value**
  - ▶ `int funzione(int)`
  - ▶ crea una copia degli argomenti della funzione
  - ▶ impossibile modificare gli argomenti
  - ▶ occupa più memoria, sconsigliato per molti dati
- ▶ Passaggio di parametri **by reference**
  - ▶ `int funzione(int&)`
  - ▶ la funzione ha accesso diretto agli argomenti
  - ▶ necessario per certe applicazioni

```
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)

```
1 #include <iostream>
2 typedef signed char schar;
3 std::istream& operator>>(std::istream&, schar&);
4 std::ostream& operator<<(std::ostream&, schar&);
5 int main(){
6     schar a;
7     std::cin >> a;
8     a += 5;
9     std::cout << a << std::endl;
10    return 0;
11 }
12 ...
```

- ▶ **typedef** per evitare conflitti
- ▶ ridefiniamo >> e << (overloading)

```
11  ...
12  std::istream& operator>>(std::istream& input, schar& x) {
13      int y;
14      input >> y;
15      x = y;
16      return input;
17  }
18  std::ostream& operator<<(std::ostream& output, schar& x)
19      {
20      output << static_cast<int>(x);
21      return output;
22  }
```



```
1  #include <iostream>
2  struct frutto {
3      int m;    //massa
4      double p; //prezzo
5  };
6  int main(){
7      frutto mela;
8      mela.m = 250;
9      mela.p = 0.28;
10     std::cout << 1e3*mela.p/mela.m << std::endl;
11     return 0;
12 }
13
```

- **struct** crea un nuovo tipo che contiene più variabili

- ▶ In OOP si creano **oggetti**
  - ▶ variabili membro
  - ▶ funzioni membro
- ▶ Una **classe** definisce un **tipo** di oggetto
- ▶ L'oggetto vero e proprio è una variabile
  - ▶ **istanza** della classe

```
2  class Rett {  
3      private:  
4          int b, h;  
5      public:  
6          Rett();  
7          Rett(int, int);  
8          int area() const;  
9          void draw() const;  
10 };  
11 ...  
12
```

- ▶ **private** introduce membri inaccessibili dall'esterno
- ▶ **public** introduce membri accessibili dall'esterno
- ▶ Funzioni pubbliche possono alterare membri privati
- ▶ **int area() const;** non modifica l'oggetto

```
2 | class Rett {  
3 |     private:  
4 |         int b, h;  
5 |     public:  
6 |         Rett();  
7 |         Rett(int, int);  
8 |         int area() const;  
9 |         void draw() const;  
10 | };  
11 | ...  
12 |
```

- ▶ **Costruttori:** `Rett()` e `Rett(int, int)`
  - ▶ vengono invocati quando si dichiara l'oggetto

```
16 ...  
17 Rett::Rett() { return; }  
18 Rett::Rett(int x, int y) {  
19     this->b = x;  
20     this->h = y;  
21     return;  
22 }  
23 ...
```

- ▶ Implementazione dei costruttori
  - ▶ all'esterno dell'oggetto
  - ▶ preceduti dal nome dell'oggetto
- ▶ **this->** indica che **b** e **h** sono membri
  - ▶ si può omettere
  - ▶ rischio di conflitto con variabili/funzioni globali

```
22 ...  
23 int Rett::area() const{  
24     return this->b*this->h;  
25 }  
26 void Rett::draw() const{  
27     for(int i = 0; i < this->h; i++) {  
28         for(int j = 0; j < this->b; j++) {  
29             std::cout << " * ";  
30         }  
31         std::cout << std::endl;  
32     }  
33     return;  
34 }  
35
```

► Implementazione delle altre funzioni

```
10 ...  
11 int main(){  
12     Rett a(8,3);  
13     a.draw();  
14     std::cout << "Area = " << a.area() << std::endl;  
15     return 0;  
16 }  
17 ...
```

- ▶ `Rett a(8,3);` chiama `Rett(int,int)`
- ▶ `Rett a;` chiama `Rett()`

```
10 | ...  
11 | int main() {  
12 |     Rett a(8,3);  
13 |     a.draw();  
14 |     std::cout << "Area = " << a.b*a.h << std::endl;  
15 |     return 0;  
16 | }  
17 | ...
```

► Errore: `b` e `h` sono privati



```
2  ...  
3  class Rett {  
4  private:  
5      int b, h;  
6  public:  
7      Rett();  
8      Rett(int, int);  
9      int area() const;  
10     void draw() const;  
11     bool operator==(const Rett&) const;  
12 };  
13 ...
```

► Overload di == per la classe Rett

```
37 ...  
38 bool Rett::operator==(const Rett& a) const{  
39     bool eq = 0;  
40     if (this->b == a.b && this->h == a.h) eq = 1;  
41     if (this->b == a.h && this->h == a.b) eq = 1;  
42     return eq;  
43 }  
44 ...
```

- ▶ Implementazione, all'esterno della classe
- ▶ Buona abitudine: passare gli argomenti by reference

```
2  ...
3  class Rett {
4  private:
5      int b, h;
6  public:
7      Rett();
8      Rett(int, int);
9      int area() const;
10     void draw() const;
11     friend bool operator==(const Rett&, const Rett&);
12 };
13 ...
```

- ▶ Alternativa: overload come funzione libera
- ▶ **friend** permette all'operatore di accedere a **private**
  - ▶ non importa dove viene scritto

```
37 ...  
38 bool operator==(const Rett& a, const Rett& b) {  
39     bool eq = 0;  
40     if (a.b == b.b && a.h == b.h) eq = 1;  
41     if (a.b == b.h && a.h == b.b) eq = 1;  
42     return eq;  
43 }  
44 ...
```

- ▶ L'operatore **non** è un membro della classe
  - ▶ per questo è necessario includerlo come **friend**
  - ▶ spesso è preferibile avere operatori membro

```
2  ...
3  class Rett {
4      private:
5          int b, h;
6      public:
7          Rett();
8          Rett(int, int);
9          int area() const;
10         void draw() const;
11         Rett& operator+=(const Rett&);
12         Rett operator+(const Rett&) const;
13         bool operator==(const Rett&);
14     };
15  ...
```

► Overload di + e +=

- + ha come output un oggetto
- += ha come output un riferimento all'oggetto

```
49  ...
50  Rett& Rett::operator+=(const Rett& a) {
51      this->b = this->b + a.b;
52      this->h = this->h + a.h;
53      return *this;
54  }
55  Rett Rett::operator+(const Rett& a) const {
56      int b = this->b + a.b;
57      int h = this->h + a.h;
58      return Rett(b, h);
59  }
60  ...
```



```
54 | ...  
55 | Rett operator+(const Rett& a, const Rett& b) {  
56 |     Rett x = a;  
57 |     x += b;  
58 |     return (x);  
59 | }  
60 | ...
```

- ▶ A volte conviene avere + come funzione libera
  - ▶ più flessibile con conversioni
- ▶ Definita in termini di +=, **non** ha bisogno di **friend**

```
2 | class Figura {  
3 | protected:  
4 |     int b, h;  
5 | public:  
6 |     Figura();  
7 |     void set_data(int, int);  
8 | };  
9 | ...
```

- ▶ Classe di base con certe caratteristiche
- ▶ **protected**, alternativa a **private/public**
  - ▶ membri **protected** sono inaccessibili dall'esterno
  - ▶ possono essere **ereditati**



```
13 | ...  
14 | Figura::Figura() { return; }  
15 | void Figura::set_data(int b, int h) {  
16 |     this->b = b;  
17 |     this->h = h;  
18 |     return;  
19 | }  
20 | ...
```

► Implementazione delle funzioni

```
8  ...  
9  class Rett : public Figura {  
10 public:  
11     Rett();  
12     int area() const;  
13 };  
14 ...
```

- ▶ Classe **derivata**
  - ▶ eredita tutti i membri **public** o **protected**
  - ▶ i membri **private** non vengono ereditati
- ▶ I costruttori devono essere ridefiniti
  - ▶ il nome del vecchio costruttore è diverso

```
19 | ...  
20 | Rett::Rett() { return; }  
21 | int Rett::area() const{  
22 |     return this->b*this->h;  
23 | }  
24 | ...
```

- ▶ Implementazione delle funzioni
- ▶ Non è necessario reimplementare `set_data`

```
23 ...  
24 int main() {  
25     Rett a;  
26     a.set_data(3,5);  
27     std::cout << "Area = " << a.area() << std::endl;  
28     return 0;  
29 }  
30 ...
```

► Utilizzo nel programma

- ▶ Una classe può ereditare da più fonti

- ▶ `class Derivata : public Fonte1, public Fonte2`

- ▶ L'eredità può modificare gli accessi

- ▶ `class Derivata : public Fonte`
  - ▶ `class Derivata : protected Fonte`
  - ▶ `class Derivata : private Fonte`

- ▶ L'eredità è alla base di una buona OOP

```
1 #include <iostream>
2 template<class T> T max(const T, const T);
3 int main(){
4     std::cout << max<int>(4,5) << std::endl;
5     return 0;
6 }
7 template<class T> T max(const T a, const T b) {
8     T c;
9     if (a >= b) { c = a; }
10    else { c = b; }
11    return c;
12 }
13
```

- Un **template** permette di creare funzioni o classi basate su classi generiche

0	1	2	3	4
15	8	3	52	27

- ▶ Un **array** è una struttura che può contenere una serie di valori, indicizzati da un numero intero
- ▶ In C++ l'indice parte sempre da 0

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

- ▶ Necessario includere l'header `<array>`
- ▶ `std::array` è un template con due parametri
  - ▶ il primo dice il tipo di dato contenuto
  - ▶ il secondo dice il numero di celle



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

- ▶ `std::size_t` è un tipo di intero senza segno
  - ▶ buona abitudine utilizzarlo nei cicli sugli array
- ▶ `.size()` restituisce il numero di elementi dell'array
- ▶ `.at(i)` permette di accedere all'*i*-esimo elemento

```
1  #include <iostream>
2  #include <array>
3  int main(){
4      const int n = 5;
5      std::array<int,n> a;
6      for(std::size_t i = 0; i < a.size(); i++) {
7          a.at(i) = 2*i;
8          std::cout << a.at(i) << std::endl;
9      }
10     return 0;
11 }
12
```

- ▶ `std::array` accetta solo parametri costanti
  - ▶ `array` statico

```
1 #include <iostream>
2 #include <vector>
3 int main(){
4     int n;
5     std::cin >> n;
6     std::vector<int> a(n);
7     for(std::size_t i = 0; i < a.size(); i++) {
8         a.at(i) = 2*i;
9         std::cout << a.at(i) << std::endl;
10    }
11    return 0;
12 }
13
```

- ▶ `std::vector` è un array dinamico
  - ▶ consumo leggermente maggiore di memoria

```
1  #include <iostream>
2  #include <vector>
3  int main() {
4      int n;
5      std::vector<int> a(10);
6      for(std::size_t i = 0; i < a.size(); i++) {
7          a.at(i) = 2*i;
8          std::cout << a.at(i) << std::endl;
9      }
10     std::cin >> n;
11     a.resize(n);
12     for(std::size_t i = 0; i < a.size(); i++) {
13         std::cout << a.at(i) << std::endl;
14     }
15     return 0;
16 }
17
```

► `.resize` ridimensiona l'array

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

► Legacy C-style array

- sconsigliato: non controlla i limiti
- `a[10]` non darebbe errore

► La libreria `<cmath>`:

- `std::pow(x, y) = xy`
- `std::sqrt(x) =  $\sqrt{x}$`
- `std::sin(x), std::cos(x), std::tan(x)`
- `std::asin(x), std::acos(x), std::atan(x)`
- `std::atan2(y, x) =  $\arctan \frac{y}{x}$  nel giusto quadrante`
- `std::exp(x) =  $e^x$`
- `std::log(x) =  $\ln x$`
- eccetera...

► La libreria `<complex>`:

- `std::complex<double>` tipo di dato
- `std::real(x) =  $\Re x$`
- `std::imag(x) =  $\Im x$`
- `std::abs(x) =  $|x|$`
- `std::arg(x) =  $\arg x$`
- overload di operazioni e funzioni

```
1 #include <iostream>
2 #include <complex>
3 int main() {
4     std::complex<double> a(-2, 0);
5     std::cout << std::sqrt(a) << std::endl;
6     return 0;
7 }
8
```

- ▶ Gestire un progetto su più file
- ▶ Librerie: `esempio.hpp`
  - ▶ dovrebbero contenere solo le dichiarazioni
  - ▶ incluse in altri file con `#include "esempio.hpp"`
  - ▶ **mai** utilizzare `using`
- ▶ Implementazione delle librerie: `esempio.cpp`
  - ▶ contengono la definizione delle funzioni o oggetti della libreria corrispondente
- ▶ File principale: `progetto.cpp`
  - ▶ contiene il `main()`



► myheader.hpp:

```
1 | #ifndef TASSONI2016
2 | #define TASSONI2016
3 | int twice(int);
4 | #endif
5 |
```

► myheader.cpp:

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

► project.hpp:

```
1 | #include <iostream>
2 | #include "myheader.hpp"
3 | int main() {
4 |     std::cout << twice(5) << std::endl;
5 | }
6 |
```

```
1 | #ifndef ETICHETTA  
2 | #define ETICHETTA  
3 | ...  
4 | #endif  
5 |
```

- ▶ **Include guard:** evita che lo stesso header sia incluso più volte se è già stato incluso tramite un altro header
- ▶ Ogni header dovrebbe avere la propria etichetta

- ▶ Compilare un progetto richiede di compilare in ordine le varie parti che lo costituiscono
- ▶ Ogni IDE gestisce la cosa in maniera diversa
- ▶ Sistemi \*nix (Linux/MacOS) supportano **Makefile**
  - ▶ estremamente flessibile nella gestione di progetti ampi
  - ▶ esempio: `CPPMakefile`

▶ Free Software

- ▶ free = libero
- ▶ free = gratuito

▶ Software di cui il codice sorgente è

- ▶ visibile a tutti: non serve fiducia
- ▶ modificabile da tutti: adattabilità

▶ Sviluppato da grandi comunità

- ▶ given enough eyeballs, all bugs are shallow
- ▶ maggiore sicurezza
- ▶ maggiore qualità

▶ Libertà

- ▶ da logiche di mercato
- ▶ da controllo esterno