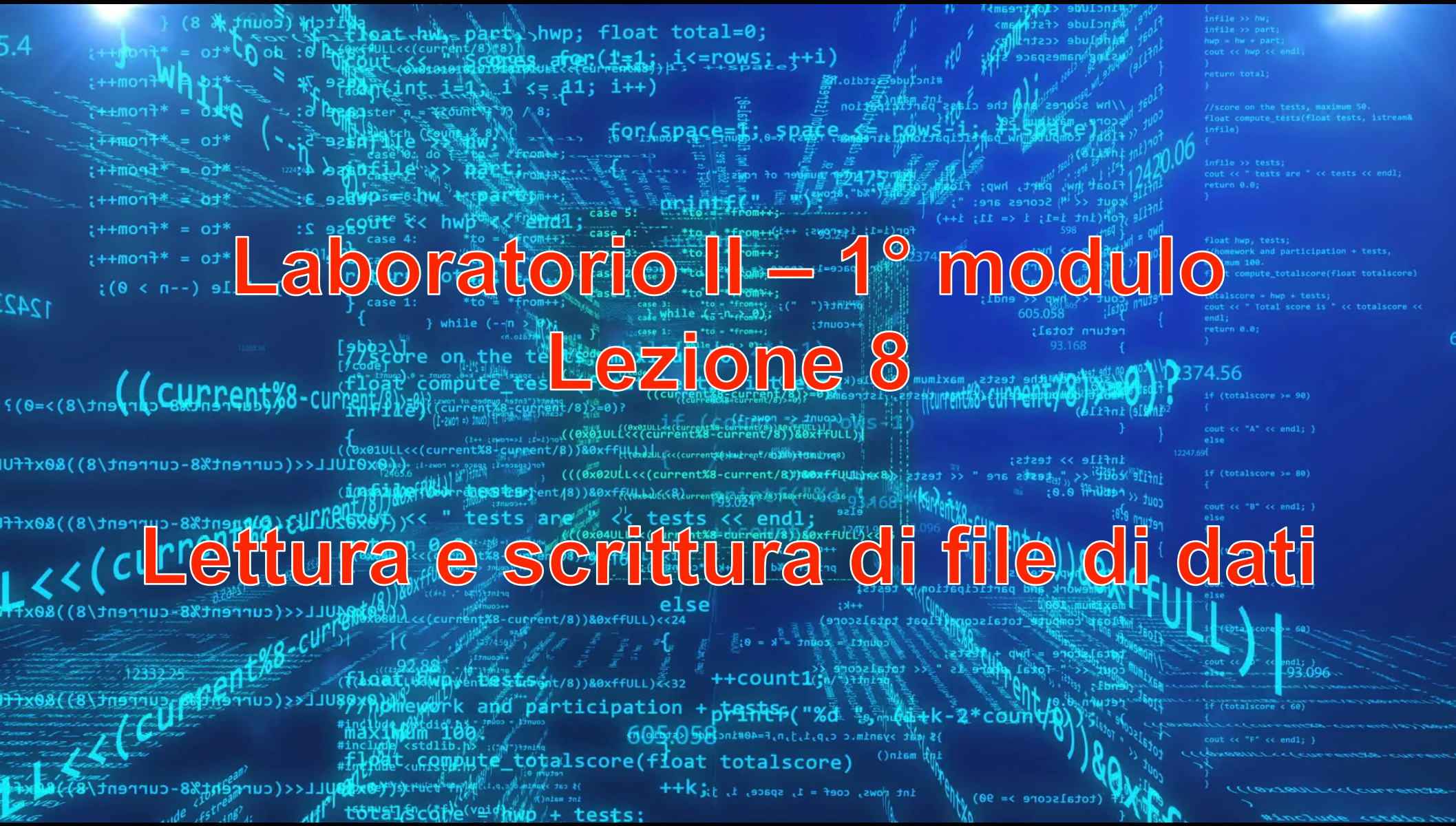
Laboratorio II – 1° modulo
Lezione 8
Lettura e scrittura di file di dati

# Indice

- Lettura e scrittura di file di dati
- Esercizi
- Approfondimenti
  - `namespace`
  - `std::stringstream`
  - `auto`

# Input/Output in C++

- Sin dal primo codice in C++ abbiamo incluso la libreria standard `iostream` che ci ha permesso di acquisire o scrivere sequenzialmente messaggi o dati attraverso il terminale

- Queste informazioni vengono fornite da riga di comando dall'utente e si perdono non appena il terminale o l'eseguibile viene chiuso

- Volendo conservare i risultati prodotti per un uso successivo, risulta necessario scrivere fisicamente (salvare) queste informazioni volatili su di un file (file di caratteri, file binario, immagine, etc…)

- I comandi di `I/O` su file sono analoghi a quelli su terminale. Bisogna includere la libreria `<fstream>` che contiene le classi per l'`I/O` su file: www.cplusplus.com/reference/fstream/fstream

# Lettura e scrittura da file

```
#include <iostream>
#include <fstream>
#include <string>
int main ()
{
  const char* filename = "File.txt";
  // Costruisce un oggetto di tipo ofstream passando
  // come parametro di input il nome del file
  std::ofstream OutFile (filename);

  int a          = 12;
  double b       = 24.5;
  char parola[] = "ciao";

  OutFile << a << "\t" << b << std::endl; // Scrive in output i valori delle variabili a e b
  OutFile << parola << std::endl;
  OutFile.close(); // Chiude il file

  int x    = 0;
  double y = 0;
  std::string word;
  // Apre il file in modalita` lettura
  std::ifstream InFile (filename);

  InFile >> x >> y; // Legge i numeri contenuti nel file e li memorizzo nelle variabili x e y
  InFile >> word;
  InFile.close();

  std::cout << "Il file " << filename <<
    " contiene i numeri: " << x << " e " << y << std::endl;
  std::cout << "e la parola: " << word << std::endl;

  return 0;
}
```

Le classi di `I/O` su file di `fstream` sono:

- `ifstream` per **input**
- `ofstream` per **output**
- `fstream` per **input/output**

Dichiara una variabile di tipo file ed apre il file di nome `filename`

Usa l'operatore `<<` per scrivere l'output sul file già aperto (come si fa su terminale con `cout`)

Usa l'operatore `>>` per leggere l'output sul file già aperto (come si fa su terminale con `cin`)

Il file può essere aperto e chiuso più volte in uno stesso programma, ma al termine del programma viene chiuso automaticamente

# Lettura e scrittura da file

**Osservazione 1:** i costruttori `ifstream` / `ofstream` / `fstream` prendono in input un puntatore a `char`:

```
int main (int argc, char** argv)
{
  // Verifica che venga passato almeno un parametro da riga di comando
  if (argc < 2)
    {
      std::cout << "Digitare il nome del file da riga di comando"
                << std::endl;
      std::cout << "\t./esempio02 File.txt" << std::endl;
      return 1;
    }

  // Usa un oggetto std::string per memorizzare il nome del file
  std::string NomeFile = argv[1];

  std::ofstream OutFile (NomeFile.c_str());
  ...
  std::ifstream InFile (argv[1]);
  ...
}
```

Usa il metodo `c_str()` per convertire l'oggetto `string` in un puntatore a `char`

Si può anche usare direttamente `argv[1]` come argomento dato che è un puntatore a `char`

# Lettura e scrittura da file

**Osservazione 2:** è possibile usare il metodo `open("NomeFile.txt", OpenMode)` di `fstream` per specificare il tipo di operazione da fare sul file attraverso *OpenMode*

```
#include <iostream>
#include <fstream>
int main ()
  {
    std::fstream OutFile;
    OutFile.open("file.txt",std::ios::out);
    OutFile << "12 24" << std::endl;
    OutFile.close();

    int a,b;
    std::fstream InFile;
    InFile.open("file.txt",std::ios::in);
    InFile >> a >> b;
    InFile.close();

    std::cout << "Ho letto: " << a << "\t" << b << std::endl;
    return 0;
}
```

**File aperto in sola scrittura**
In `ofstream` l'*OpenMode* è `std::ios::out` per default

`std::ios::app` si usa per scrivere su un file in modalità *append* (cioè aggiunge righe a quelle già esistenti)

**File aperto in sola lettura**
In `ifstream` l'*OpenMode* è `std::ios::in` per default

# Se un file ha molte righe...

```
#include <iostream>
#include <fstream>
#include <string>

int main (int argc, char** argv)
{
  // Verifica che venga passato almeno un parametro da riga di comando
  if (argc < 2)
    {
      std::cout << "Digitare il nome del file da riga di comando"
                << std::endl;
      std::cout << "\t./esempio03 File.txt" << std::endl;
      return 1;
    }
  std::string NomeFile = argv[1];
  std::ofstream OutFile (NomeFile.c_str());
  // Scrittura di piu` righe (std::endl interrompe la riga)
  for (int i = 0; i < 10; i++)
    {
      OutFile << i*2 << "\t" << i*2+1 << std::endl;
    }
  OutFile.close();
...
```

# Se un file ha molte righe...

```
...
  int x      = 0;
  int y      = 0;
  int Nrighe = 0;
  std::ifstream InFile (NomeFile.c_str());

  // Imposta un loop infinito interrotto con l'istruzione break
  while (true)
    {
      InFile >> x >> y;

      // Il metodo eof() restituisce true alla fine del file
      if (InFile.eof() == true)
        break;

      std::cout << x << "\t" << y << std::endl;
      Nrighe++;
    }

  InFile.close();
  std::cout << "Il file " << NomeFile << " contiene "
            << Nrighe << " righe" << std::endl;
  return 0;
}
```

Esempio di contenuto del file

| | |
|---|---|
| 0 | 1 |
| 2 | 3 |
| 4 | 5 |
| ... | ... |

# Se un file ha molte righe...

**N.B.:** questo codice non funziona (stampa 2 volte l'ultima riga del file):

```
...
while (InFile.eof() == false)
  {
    InFile >> x >> y;
    std::cout << x << "\t" << y << std::endl;
    Nrighe++;
  }
...
```

Il motivo è che per far diventare `InFile.eof() == true` è necessario effettuare l'operazione di lettura anche dopo aver letto l'ultima riga del file. E` un po' come se il file avesse `N+1` righe (dove `N` è il numero di coppie `x,y` salvate nel file), cioè l'ultima vera riga del file, la `N+1`-esima, contenente il carattere di "`EOF`" (i.e. `End Of File`) Dopo ogni riga del file letta, e prima di manipolare i dati (e.g. stamparli a schermo), è necessario quindi effettuare il controllo di `eof() == true`

# Tips & tricks

È buona norma inserire dei **controlli** che verificano l'integrità del file aperto. Questo vale sia quando si effettua una **lettura** da file sia quando si effettua una **scrittura** su file

```
std::ifstream InFile (filename.c_str());
// Se il file non è presente nella cartella o non e` possibile aprirlo
// stampa un messaggio di errore
if (InFile.good() == 0)
{
  std::cout << "Errore! Non è possibile aprire il file" << std::endl;
  return 1;
}
```

Esempio di lettura di dati e salvataggio in `std::vector`

```
std::vector<int> vec1;
std::vector<int> vec2;
while (true)
{
  InFile >> x >> y;
  if (InFile.eof() == true)
    break;
  vec1.push_back(x);
  vec2.push_back(y);
  Nrighe++;
}
```

**N.B.**: non serve conoscere a priori il numero dei dati contenuti nel file

# Esercizi

**Esercizio 1:** Scrivere un codice che legga i dati contenuti nel file `dati.txt` ed inserirli in due `std::vector<double>`. Al termine della lettura del file di input, impostare un ciclo `for` per rileggere i numeri salvati e calcolare la media e la deviazione standard dei dati contenuti nei due `std::vector`. Stampare i risultati in un file di output. Usare `argc` e `argv` per inserire da terminale i nomi dei file di input e di output

- Leggere i dati implementando una funzione con il seguente prototipo (restituisce `false` se la lettura non è andata a buon fine, altrimenti `true`)

```
bool readData(std::string fileName,
std::vector<double>& valVec, std::vector<double>&
errVec)
```

**Esercizio 2:** Creare la classe dei vettori geometrici (2D) nel piano. Implementare le operazioni con i vettori (somma, sottrazione, prodotto, etc...) secondo l'header file fornito (`vett2d.h`)

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

# Approfondimenti

- `namespace`
- `std::stringstream`
- `auto`

# namespace

I **namespace** permettono di raggruppare varie entità sotto un unico un nome:

```
namespace nomespazio
  {
    int a;
    double b;
  }
```

- Per utilizzarli: `nomespazio::a`
  `nomespazio::b`
- Proprio come il `namespace` "`std`": `std::cout << "ciao" << std::endl;`
- L'uso dei `namespace` permette di evitare conflitti tra nomi di identificatori (variabili, funzioni, strutture, etc…), soprattutto in codici di grandi dimensioni e complessi, permettendo così di scrivere codici più sicuri e di più facile *debugging*
- Si può comunque omettere il riferimento al `namespace` attaverso l'istruzione `using namespace`, da riportare prima dell'inizio del `main`, ad esempio: **`using namespace std;`**

# std::stringstream

Gli oggetti di tipo `stringstream` hanno a disposizione dei metodi molto utili per manipolare flussi di dati, potendo gestire sia numeri che parole in maniera analoga ai metodi `std::cin` e `std::cout`

```
#include <iostream>
#include <sstream>

using namespace std;
int main ()
{
  stringstream ss;
  int anno    = 1923;
  string nome = "John";

  // Posso unire numeri e stringhe in un unico stream
  ss << nome << " nacque nel " << anno;
  // Conversione sstream -> string
  string frase = ss.str();
  cout << frase << endl;

  return 0;
}
```

**N.B.:** per svuotare una `stringstream` è necessario utilizzare la coppia di metodi:

```
ss.clear();
ss.str("");
```

Si scrive nella variabile `stringstream` con l'operatore `<<` (come con `std::cout`)

# Lettura file con stringstream

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
using namespace std;
int main(int argc, char** argv)
{
  string filename = argv[1];
  ifstream InFile (filename.c_str());
  // Metodo good() per verificare che sia possibile leggere il file
  if (InFile.good() == false)
    {
      cout << "Errore! Non è possibile aprire il file " << filename << endl;
      return 1;
    }
  // Usa il metodo getline (istream& is, string& str)
  // per leggere il file una riga alla volta
  // (restituisce falso alla fine del file)
  string line, parola;
  stringstream ss;
  int nRighe = 0;
  while (getline(InFile, line))
    {
      nRighe++;
      cout << "\nRiga " << nRighe << ": " << line << endl;
      cout << "Selezione parole pari: ";

      int nParole = 0;
      ss << line; // Riempie lo stringstream con la riga
      while (ss >> parola) // Legge parola per parola
            {
              nParole ++ ;
              if ((nParole%2) == 0)
                cout << parola << " ";
              else continue;
            }
      cout << endl;
      ss.clear(); ss.str(""); // Svuota lo stringstream
    }
  return 0;
}
```

Una variabile `stringstream` può essere usata per `I/O` da file

`getline` prende in input un oggetto `ifstream` (file da leggere) e un oggetto `string`, in cui memorizzare la riga del file. Restituisce **`false`** alla fine del file

Con l'operatore **>>** viene letto lo stream come si farebbe con `std::cin`

In questo esempio abbiamo si è deciso di selezionare solo le parole che occupano una posizione pari all'interno della riga

# L'attributo `auto`

L'attributo `auto` (introdotto dal `C++11` in poi) consente di dichiarare delle variabili di tipo non specificato perché il tipo verrà automaticamente dedotto a *compile time*. Inoltre può essere usato nelle funzioni che fanno uso di `return` per "alleggerire" la sintassi:

```
auto a = 1 + 2; // Ad a viene assegnato il tipo int
auto b = 1 + 1.2; // A b viene assegnato il tipo double
auto x; // Fornisce errore
for (auto i=0; i<10; i++) // Ad i viene assegnato il tipo int
auto myFunction(...) -> decltype ( ... )
{
   ...
   return val;
}
```

Notare anche questo nuovo elemento sintattico

Nel `C++14` e `C++17` l'uso dell'attributo `auto` è stato esteso ad altri casi

**Esercizio:** Usare nell'esercizio 1 l'attributo `auto` dove possibile

# L'attributo auto

Esempio di uso di **auto** per identificare il tipo associato al return:

```
#include <iostream>
#define N 100
int makeObject()
{
  return N;
}
auto testReturn() -> decltype ( makeObject() )
{
  auto val = makeObject();
  std::cout << "I use the new type: " << val << std::endl;
  return val;
}
int main ()
{
  testReturn ();
  return 0;
}
```

Notare anche questo elemento sintattico: **decltype (...)** consente di ottenere il tipo di ciò che sta tra parentesi

Tramite l'uso congiunto di **auto** e **decltype** non si è dovuto specificare il tipo di `val` e nemmeno del `return`

# L'attributo `auto`

Inoltre la potenza di **`auto`** la si apprezza anche nell'uso dei cicli `for` dove non è più necessario specificare il tipo dell'elemento dell'array su cui si itera. Ecco un esempio di **"iteratore universale"** che fa uso dell'attributo **`auto`** e dei template:

```
...
template<typename T>
void myGenericIterator (T& myIterable) {
  for (auto ele : myIterable)
    std::cout << ele << std::endl;
}
int main()
{
  std::string myArray[] = {"Giulia", "Alessandro", "Pietro"};
  std::vector<std::string> myVec;
  myVec.push_back(myArray[0]);
  myVec.push_back(myArray[1]);
  myVec.push_back(myArray[2]);
  myGenericIterator(myVec);
  std::cout << std::endl;
  myGenericIterator(myArray);
  return 0;
}
```

Notare che è stata omessa la specifica del tipo: `function_name<type>(...)` perché risulta superflua quando, come in questo caso, il tipo può essere dedotto a *compile time*. Si chiama *template argument deduction*