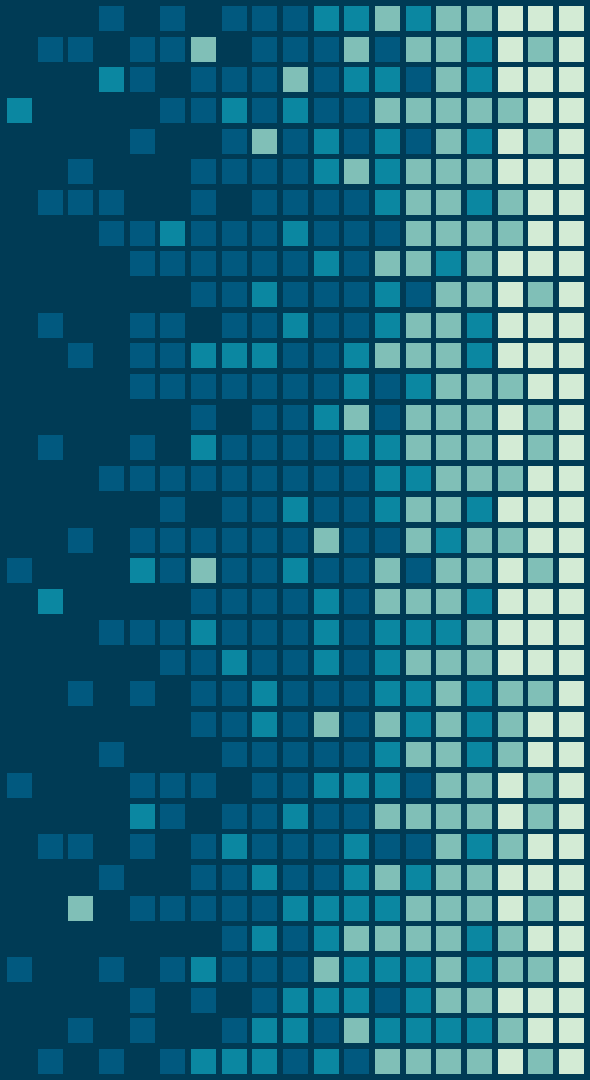
C++ from zero to hero
Parte 3



1.

Algorithms

...more on STL

80 anni di informatica...

...hanno prodotto una raffinata conoscenza di tanti algoritmi. Perché reinventarli ogni volta, se possiamo accedere alle conoscenze di “mostri sacri” come Von Neumann, Tony Hoare e altri richiamando le loro conoscenze tramite funzioni?

Questo è lo spirito di `<algorithm>`, essenziale per una programmazione efficiente e “parlante” (utile, ad esempio, nella competitive programming).

Algoritmi: principio base

- Lavorano su range di iteratori, come abbiamo accennato la precedente lezione:

```
algoritmo(begin, end [,...])
```

Alcuni esempi base

- `swap(a,b)`
- `sort(begin(v), end(v))`
- `is_sorted(begin(v), end(v))`
- `reverse(begin(v), end(v))`
- `max(begin(v), end(v))`
- `min(begin(v), end(v))`
- `accumulate(begin(v), end(v), 0) // <numeric>`

Alcuni esempi più arditi

- `copy(begin(v), end(v), back_inserter(dest));`
- `copy(v.begin(), v.end(), v1.begin());`
- `fill(begin(v), end(v), valore));`
- `remove(begin(v), end(v), valore));`
- `replace(begin(v), end(v), val_vecchio, val_nuovo));`
- `find(begin(v), end(v), valore)); // torna iteratore`
- `binary_search(begin(v), end(v), valore)`
- `count(begin(v), end(v), valore)`

Esempio: for_each

```
#include <algorithm>

void show(int n) { cout << n << " ";}

vector<int> v{ 12, 3, 17, 8 };
for_each (begin(v), end(v), show);
```

Esempio: find

```
int x; vector<int> v{ 12,3,17,8,34,56,9 };
cout << ”Inserisci valore da cercare”;
cin >> x;
auto where = find(begin(v), end(v), 8);
if (where != end(v))
   cout<<”trovato!: ”<< distance(begin(v), iter);
else
   cout<<”Non trovato!”:
```

Esempio: accumulate

```
vector<int> v{ 1, 4, 5, 2, 7 };
cout << accumulate(begin(v), end(v), 0); // 19

cout << accumulate(
   begin(v), end(v), 0, multiplies<>{}); // 280
```

Lambda Expressions: in pillole

- Molti algoritmi hanno dei “customization point” passabili tramite “funzioni” (o più in generale attraverso dei **function objects**)

- Un function object è un tipo che implementa il “call operator” (`operator()()`)

Lambda Expressions: in pillole

```
struct logger {
    void operator()(const string& message) {
       cout << message << "\n";
};
logger l;
l("hello"); // chiama il "call operator"
vector<string> v = {"ciao", "come", "stai?"};
for_each(begin(v), end(v), l);
```

Lambda Expressions: in pillole

- In un caso come quello di prima, è sconveniente creare un oggetto ogni volta, specialmente se il suo ambito è in qualche maniera "usa e getta"
- Ci vengono in aiuto la **lambda expressions** (spesso chiamate solo **lambda**), ovvero dei **generatori di function object portatili**

Lambda Expressions: in pillole

```
vector<string> v = {“ciao”, “come”, “stai?”};
for_each(begin(v), end(v), [](const string& s){
    cout << s << “\n”;
});
```

Lambda Expressions: in pillole

- La potenza delle lambda expression è che generano un oggetto che può essere dotato di **stato**

```
auto counter = 0;
for_each(begin(v), end(v), [&counter](const string& s) {
    ++counter;
});
```

Lambda Expressions: in pillole

- È possibile "catturare" le variabili dello scope per **valore** (copia) o per **reference**:

```
[&byRef, byValue] () { … }
```

- È possibile definirne di nuove:

```
[pippo = 10] (string& ) { ... }
```

- È possibile usare auto nella lista dei parametri:

```
[](auto& param1, auto param2) { . . . }
```

2.
Programmazione orientata agli oggetti

Oggetto

In informatica, la programmazione orientata agli oggetti (in inglese object-oriented programming, in acronimo OOP) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi.

Oggetto come struttura dati

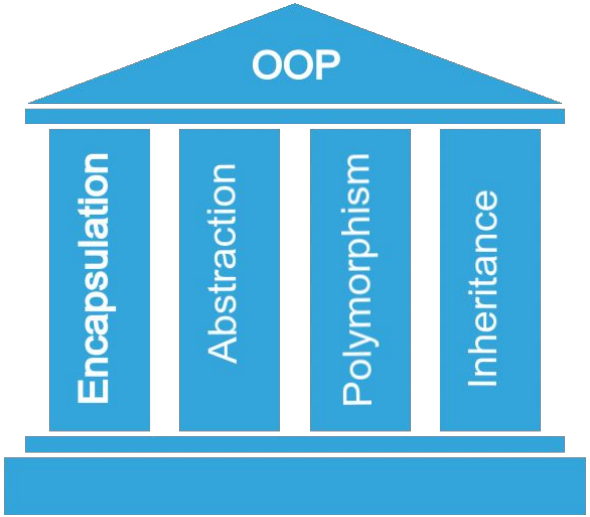
In prima approssimazione, possiamo vedere un oggetto come un aggregato di variabili trattate con un'entità unica. . Per esempio possiamo definire una struttura nel modo seguente. Per accedere ai membri dell'oggetto si usa l'operatore . (o -> nel caso di puntatori)

Questa visione è tuttavia riduttiva.

```
typedef struct complesso {
    float reale;
    float immag;
};
complesso c = {1.2, 2.3};
cout  << c.reale;
```

Paradigma di programmazione

In questa sede non discuteremo del modello OOP in sé, ma vedremo come il C++ applica i suoi principi:



Incapsulamento

Il principio ci dice che tutto quanto fa parte di un oggetto deve essere compreso nell'oggetto stesso. In particolare, sono inclusi tutti gli attributi (=variabili) e tutti i metodi (=funzioni) necessari per utilizzarlo.

La parola chiave per farlo è **class**, che di fatto definisce un nuovo tipo di dato.

```
class Punto {
  public:
     float x;
     float y;
    float distanza(Punto p);
};

float Punto::distanza(Punto p) {
    return
sqrt((this->x-p.x)*(this->x-p.x) +
(this->y-p.y)*(this->y-p.y));
};
```

Astrazione

Nascondere tutto quanto non strettamente necessario

Questo porta alla creazione di alcune classi di visibilità (**public, private, protected**) che permettono di nascondere i dettagli implementativi e proteggere gli oggetti da usi impropri.

Implica la creazione di metodi specializzati per la manipolazione dei dati (*getter, setter, costruttori, distruttori*)

E' possibile "violare" queste regole tramite la parola chiave **friend**.

Astrazione: esempio classico

La funzione area è pubblica:

Radius è privato, ma si può controllarne il valore con una funzione pubblica.

```
class Cerchio
{    private:
         double radius;
     public:
         double  arearea()
         {   return 3.14*radius*radius;
         }
         double  getradius()
         {   return radius;
         }

};
```

Ereditarietà

Meccanismo che permette di espandere le funzionalità riusando il codice e le strutture esistenti.

I figli possono accedere ai metodi pubblici del padre ed anche a quelli protetti.

E’ possibile l’eredità multipla.

Ereditarietà (esempio)

Punto3d è formato da x,y,z e dal metodo distanza.

```
class Punto {
  public:
      float x;
      float y;
     float distanza(Punto p);
};
class Punto3d::Punto {
  public:
      float z;
};
```

Polimorfismo

Preso dal greco (*molte-forme*), l'applicazione del principio permette di gestire più funzioni con lo stesso nome (cosa vietatissima in C, per esempio).

La specifica versione che sarà eseguita dipende dal contesto (parametri, ereditarietà e altro)

Il polimorfismo si declina in diverse varianti.

Polimorfismo 1 - Overloading

Permette di definire più “versioni” di un metodo, utilizzando diverse “firme” (*signature*) di metodi.

I parametri ed eventualmente i qualificatori (e.g. *const*) determinano quale metodo viene eseguito.

void foo(int i)

void foo(string s)

Polimorfismo 2 - Overriding

Permette di ridefinire (fornire una diversa implementazione) una funzione di una classe base nelle classi figlie.

Tale intenzione va esplicitata dalla parola chiave **virtual**.

(In altri linguaggi l'override è automatico)

Overriding: esempio

```
class Logger {
public:
    virtual void Log(const string& message)
    {
        cout << message << endl;
    }
};
```


Overriding

```
class FileLogger : public Logger
public:
    FileLogger() : mFile("log.txt") { }
    void Log(const string& message) override
    {
      mFile << message << endl;
    }
private:
  ofstream mFile;
};
```

Interfaccia

Rappresenta un “contratto che una classe si impegna a rispettare”, ovvero una serie di funzioni che implementa.

L’interfaccia è il concetto più importante della programmazione orientata agli oggetti.

Interfaccia

In C++ il concetto di interfaccia non è implementato a livello di linguaggio ma si può convenzionalmente associare al caso di una classe contenente solo funzioni **virtuali pure** (e.g. = 0).

Altri linguaggi (e.g. Java, C#) hanno introdotto una keyword specifica (**interface** appunto) ed una sintassi ad-hoc.

Interfaccia in C#

```
public interface ILogger {
   void Log(string message);
};

public class ConsoleLogger : ILogger {
   public void Log(string message) {
     Console.WriteLine(message);
   }
}
```

Interfaccia in Java

```
public interface ILogger {
   void Log(string message);
};

public class ConsoleLogger  implements ILogger {
   public void Log(string message) {
     System.out.println(message);
   }
}
```

Interfaccia in C++

```
class ILogger {
public:
   virtual ~ILogger() = default;
   virtual void Log(const string& message)  = 0;
};

class ConsoleLogger : public ILogger {
public:
  void Log(const string& message)  override {
    cout << message << "\n";
  }
};
```

Interfaccia

Nell'OOP, l'interfaccia esprime al meglio il principio di **sostituibilità.**

Nel nostro semplice esempio del logger, un "cliente" dell'interfaccia logger può utilizzare qualsiasi istanza che la implementa senza cambiare niente del proprio codice.

Una applicazione di uno dei principi SOLID della progettazione (Open/Closed principle)

Interfaccia: sostituibilità

```
// client
void UsaLogger(ILogger& logger) {
   logger.Log(“hello world”);
}

...
ConsoleLogger logger;
UsaLogger(logger);
//...
FileLogger logger;
UsaLogger(logger);
```

Attenzione alla copia...

```
ConsoleLogger consoleLogger;
ILogger logger = consoleLogger; // slicing!

ILogger& logger = consoleLogger; // ok
```

Come le posso mettere in un vector?

```
vector<ILogger> v ; ???
```

Un po' fuori dallo scope del corso "base", se siete interessati lo vediamo nei contenuti bonus...

Spoiler per gli arditi:

```
// una possibile soluzione
vector<unique_ptr<ILogger>> v;
```

VOTING TIME!

E’ tutto!
Grazie per aver partecipato!