Funtori

Un funtore è un oggetto di una classe che può essere trattato come fosse una funzione (o un puntatore a funzione):

```
FunctorClass fun;
fun(1,4,5);
```

È possibile fare ciò mediante l'overloading di `operator()`, l'operatore di "chiamata di funzione": può avere un qualsiasi numero di parametri di qualsiasi tipo e ritornare qualsiasi tipo. Quando si invoca `operator()` su un oggetto, si può quindi pensare di "invocare" quel funtore.

```
class FunctorClass {
private:
  int x;
public:
 FunctorClass(int n): x(n) {}
 int operator() (int y) const {return x+y;}
};

int main() {
  FunctorClass sommaCinque(5);
  cout << sommaCinque(6); // stampa 11
}
```

```
class MoltiplicaPer {
private:
  int factor;
public:
  MoltiplicaPer(int x): factor(x) {}
  int operator() (int y) const {return factor*y;}
};

int main() {
  vector<int> v;
  v.push_back(1); v.push_back(2); v.push_back(3);
  cout << v[0] << " " << v[1] << " " << v[2]; // stampa 1 2 3
  std::transform (v.begin(), v.end(), v.begin(), MoltiplicaPer(2));
  cout << v[0] << " " << v[1] << " " << v[2]; // stampa 2 4 6
}
```

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first, InputIterator last,
                            OutputIterator result, UnaryOperation op);

/*
Applica op ad ogni elemento in [first,last) e memorizza il valore ritornato da ogni
applicazione di op nel segmento di contenitore che inizia da result.
Equivalente al seguente codice:
*/

template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first, InputIterator last,
                            OutputIterator result, UnaryOperation op) {
  while (first != last) {
    *result = op(*first);
    ++result; ++first;
  }
  return result;
}
```

```
class UgualeA {
private:
  int number;
public:
  UgualeA(int n): number(n) {}
  bool operator() (int x) const {return x==number;}
};

// template di funzione, con parametro di tipo "funtore" int -> bool
template<class Functor>
vector<int> find_matching(const vector<int>& v, Functor pred) {
  vector<int> ret;
  for(vector<int>::const_iterator it = v.begin(); it<v.end(); ++it)
    if( pred(*it) ) ret.push_back(*it); // deve essere disponibile bool operator()(int)
  return ret;
}

int main() {
  vector<int> w;
  w.push_back(1); w.push_back(5); w.push_back(1); w.push_back(3);
  vector<int> r = find_matching(w, UgualeA(1));
  for(int i=0; i<r.size(); ++i) cout << r[i] << " ";
  // stampa 1 1
};
```

Altro esempio d'uso di funtori con `std::for_each`

```
class Functor {
  int fattore;
public:
  Functor(int m=1): fattore(m) {}
  void operator() (int x) const {cout << fattore*x << " ";}
};

void fun1(const vector<int>& v) {
  Functor f(2); // funtore int -> void
  std::for_each(v.begin(), v.end(), f);
}
```

`std::for_each(InputIterator first, InputIterator last, UnaryFunction f)`

è un template di funzione di STL dichiarata in **`<algorithm>`**

```
class Functor {
  int fattore;
public:
  Functor(int m=1): fattore(m) {}
  void operator() (int x) const {cout << fattore*x << " ";}
};

void fun1(const vector<int>& v) {
  Functor f(2); // funtore int -> void
  std::for_each(v.begin(), v.end(), f);
}
```

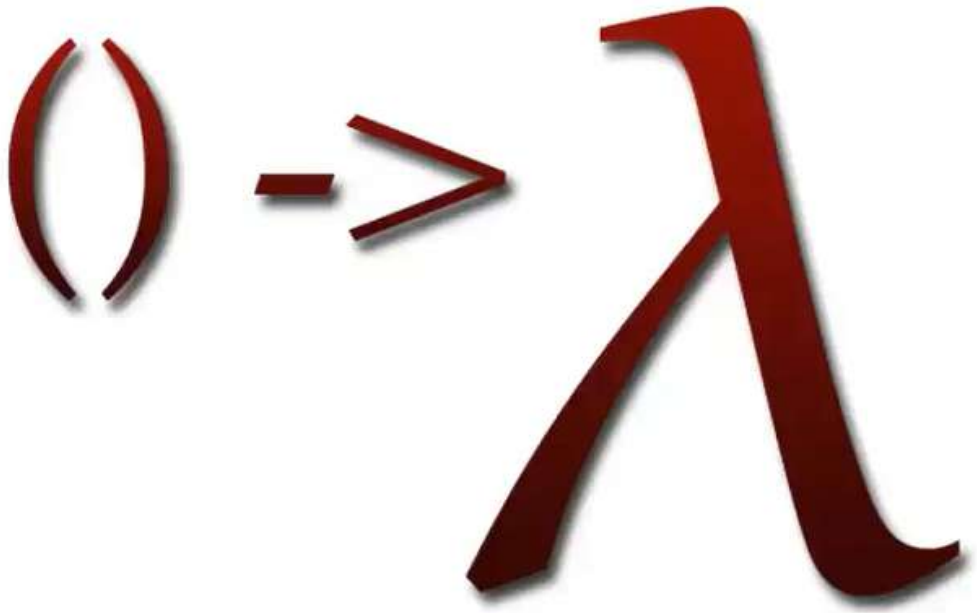
```
int fattore=2;

void fun2(const vector<int>& v) {
  std::for_each(v.begin(), v.end(),[fattore](int x)
                                   {cout << fattore*x << " ";})
}
```

Con lambda espressione

Lambda espressioni o closures

Alias: funtori anonimi



Funzione anonima locale

`[capture list] (lista parametri) ->return-type { corpo }`

`(lista parametri)` e `->return-type` sono opzionali

```
[] ->int {return 3*3;}            // lista vuota di parametri
[](int x, int y) {return x+y;} // tipo di ritorno implicito: int
[](int x, int y) ->int {return x+y;} // tipo di ritorno esplicito: int
[](int& x) {++x;}                 // tipo di ritorno implicito: void
[](int& x) ->void {++x;}          // tipo di ritorno esplicito: void
```

Closure

`[capture list]` elenca la lista delle variabili della closure, cioè variabili all'esterno della lambda espressione usate come l-valore (lettura e scrittura) o r-valore (sola lettura) dalla lambda espressione.

Closure

`[capture list]` elenca la lista delle variabili della closure, cioè variabili all'esterno della lambda espressione usate come l-valore (lettura e scrittura) o r-valore (sola lettura) dalla lambda espressione.

Closure (computer programming)

From Wikipedia, the free encyclopedia

In programming languages, **closures** (also **lexical closures** or **function closures**) are techniques for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment:[1] a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] A closure—unlike a plain function—allows the function to access those *captured variables* through the closure's copies of their values or references, even when the function is invoked outside their scope.

Closure

`[capture list]` elenca la lista delle variabili della closure, cioè variabili all'esterno della lambda espressione usate come l-valore (lettura e scrittura) o r-valore (sola lettura) dalla lambda espressione.

```
[]       \\ nessuna variabile esterna catturata
[x, &y]  \\ x catturata per valore, y per riferimento
[&]      \\ tutte le variabili esterne catturate per riferimento
[=]      \\ tutte le variabili esterne catturate per valore
[&, x]   \\ tutte le variabili per riferimento, tranne x per valore
```

Esempio

```
#include<algorithm>     // dichiarazione del template for_each
for_each(InputIterator first, InputIterator last, UnaryFunction f)

// funzione che ritorna true se e solo se c è "maiuscola"
bool is_upper(char c);

int main() {
  char* s = "Hello World";
  int UppercaseNum = 0; // nella closure della lambda espressione
  std::for_each(s, s+sizeof(s), [&UppercaseNum] (char c) {
                                  if (is_upper(c)) UppercaseNum++;
                                }
                );

 cout<< UppercaseNum << " lettere maiuscole in: " << s << endl;
}
```

`this` può essere catturato solo per valore

```
class C {
  ...
  int f() const {...}

  int m(const vector<int>& v) const {
    int totale = 0;
    int a = someClass::getSomeIntValue();
    std::for_each(v.begin(), v.end(), [&totale, a, this](int x) {
      totale += x * a * this->f();
    } );
    return totale;
  }
};
```

Esempio: capture di un parametro

```
void fun(const std::vector<int>& v, int fattore) {
  std::for_each(v.begin(), v.end(),
     [fattore](int x) {std::cout << fattore*x << " ";} )
}
```

```
void fun(std::vector<double>& v, double epsilon){
  std::transform(v.begin(), v.end(), v.begin(),
       [epsilon](double d) -> double {
         if (d < epsilon) { return 0; }
         else { return d; }
       }
  );
}
```

Esempio

```
class RubricaEmail {
private:
  vector<string> rub; // rubrica di email
public:
  // un template di metodo permette di istanziare sia a funtori che a lambdas
  // Functor con parametro const string& che ritorna un bool
  template<class Functor> // funtore const string& -> bool
  vector<string> trovaIndirizzi(Functor test) const {
    vector<string> ris;
    for(auto it = rub.begin(); it != rub.end(); ++it)
      if (test(*it)) ris.push_back(*it);
    return ris;
  }
};

vector<string> trovaIndirizziGmail(const RubricaEmail& r) {
  return r.trovaIndirizzi(
       [] (const string& email) {
                                  return email.std::find("@gmail.com") != string::npos;
                                 }
                          );
}

vector<string> trovaIndirizziConMatch(const RubricaEmail& r, const string& match) {
  return r.trovaIndirizzi(
       [match] (const string& email) {
                                       return email.std::find(match) != string::npos;
                                      }
                          );
}
```