Il C++ include delle funzioni generiche come std::for_each e std::transform, sempre utili ma non nel caso in cui il funtore sia unico ad una certa funzione (ricordando che il funtore non è altro che una classe con dei parametri che "assomiglia" ad una funzione).

Un funtore (o oggetto funzione) è una classe C++ che agisce come una funzione. I funtori vengono chiamati utilizzando la stessa vecchia sintassi di chiamata di funzione. Per creare un funtore, creiamo un oggetto che sovraccarica l'operatore(). Vediamo ad esempio:

```
using namespace std;

// A Functor
class increment
{
private:
   int num;
public:
   increment(int n) : num(n) {  }

   // This operator overloading enables calling
   // operator function () on objects of increment
   int operator () (int arr_num) const {
      return num + arr_num;
   }
};

// Driver code
int main()
{
   int arr[] = {1, 2, 3, 4, 5};
   int n = sizeof(arr)/sizeof(arr[0]);
   int to_add = 5;

   transform(arr, arr+n, arr, increment(to_add));

   for (int i=0; i<n; i++)
      cout << arr[i] << " ";
}
```

Output: 6 7 8 9 10

Quindi, qui, Increment è un funtore, una classe c ++ che funge da funzione.

La linea: `transform(arr, arr+n, arr, increment(to_add));`

equivale a scrivere:

`increment obj(to_add);` e `transform(arr, arr+n, arr, obj);`

Ora, un piccolo esempio in cui si crea un funtore apposito per un contesto specifico:

```
#include <algorithm>
#include <vector>

namespace {
 struct f {
  void operator()(int) {
   // do something
  }
 };
}

void func(std::vector<int>& v) {
 f f;
 std::for_each(v.begin(), v.end(), f);
}
```

Si noti come la struttura *f* venga utilizzata solamente per lo scopo di invocare il funtore in questo contesto, risultando quindi uno spreco.

Il C++11 introduce le lambda funzioni, per scrivere piccoli pezzi di codice riutilizzabili in vari contesti e quindi evitare di doversi definire una struttura apposita o un funtore apposito per eseguire una certa cosa. Ad esempio:

```
void func3(std::vector<int>& v) {
 std::for_each(v.begin(), v.end(), [](int) { /* do something here*/ });
}
```

Hanno questa sintassi:

```
[ capture clause ] (parameters) -> return-type
{
  definition of method
}
```

Un'espressione lambda può avere più potenza di una funzione ordinaria avendo accesso alle variabili dall'ambito che la racchiude.
Possiamo acquisire variabili esterne dall'ambito della capture clause in tre modi:
- Cattura per riferimento
- Acquisizione per valore
- Acquisizione in entrambi i modi(acquisizione mista)

Sintassi utilizzata per catturare le variabili:
[&] : cattura tutte le variabili esterne per riferimento
[=] : cattura tutte le variabili esterne per valore
[a, &b] : cattura a per valore e b per riferimento
Un lambda con captura clause vuota [] può accedere solo alle variabili locali.

Tipi di ritorno

Nei casi semplici, il compilatore deduce automaticamente il tipo di ritorno, ad esempio:

```
void func4(std::vector<double>& v) {
  std::transform(v.begin(), v.end(), v.begin(),
              [](double d) { return d < 0.00001 ? 0 : d; }
              );
}
```

tuttavia esistono alcuni casi in cui il compilatore non riesce a dedurle da solo, come ad esempio:

```
void func4(std::vector<double>& v) {
   std::transform(v.begin(), v.end(), v.begin(),
      [](double d) {
         if (d < 0.0001) {
            return 0;
         } else {
            return d;
         }
      });
}
```

Per risolvere questo problema è possibile specificare esplicitamente il tipo di ritorno per una funzione lambda, utilizzando -> T:

```
void func4(std::vector<double>& v) {
   std::transform(v.begin(), v.end(), v.begin(),
      [](double d) -> double {
         if (d < 0.0001) {
            return 0;
         } else {
            return d;
         }
      });
}
```

Cattura delle variabili

Finora non abbiamo usato nient'altro che ciò che è stato passato alla lambda al suo interno, ma possiamo anche usare altre variabili, all'interno della lambda. Se si desidera accedere ad altre variabili è possibile utilizzare la clausola di acquisizione ([] dell'espressione), che finora non è stata utilizzata in questi esempi, ad esempio:

```
void func5(std::vector<double>& v, const double& epsilon) {
   std::transform(v.begin(), v.end(), v.begin(),
      [epsilon](double d) -> double {
         if (d < epsilon) {
            return 0;
```

```
        } else {
            return d;
        }
    });
}
```