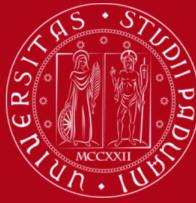


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Reference

Stefano Ghidoni



Agenda

- Reference
- Const reference
- Passaggio per copia vs passaggio per reference
- Questioni di efficienza



Reference

- Le reference possono essere usate anche in altri contesti

```
int i = 7;

int& r = i;      // reference a i
r = 9;           // i ora contiene 9
i = 10;          // i ora contiene 10

cout << r << ' ' << i << '\n';      // stampa: 10 10
```

- i e r sono alias!
 - Anche in scrittura

La reference, una volta creata, si aggancia alla variabile e diventa equivalente, diventano alias in lettura e scrittura. In più, la reference non occupa spazio addizionale.



Reference

- Considerate questo esempio:

```
vector< vector<double> > v;  
  
// comodo se dobbiamo accedere in lettura a quel valore  
// molte volte  
double val = v[f(x)][g(y)];  
  
// e se dobbiamo accedere in scrittura?
```



Reference

- Considerate questo esempio:

```
vector< vector<double> > v;  
  
// comodo se dobbiamo accedere in lettura a quel valore  
// molte volte  
double val = v[f(x)][g(y)];  
  
// e se dobbiamo accedere in scrittura?  
double& var = v[f(x)][g(y)];  
  
// ora possiamo scrivere su var!  
var = var / 2 + sqrt(var);
```



Passaggio per riferimento

- Il passaggio per riferimento fornisce accesso in lettura e scrittura!
 - Protezione dei dati?
- Limitare l'accesso alla sola lettura può essere molto importante
- Per preservarlo: passaggio per riferimento costante

es: la funzione print non dovrebbe avere diritto di scrivere, tuttavia trasformando la funzione passando da copia a riferimento, a print viene dato il permesso di scrivere. Per risolvere questo problema si usa il passaggio per riferimento costante



Passaggio per riferimento const

```
void print(const vector<double>& v)
{
    cout << "{";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1)
            cout << ", ";
    }
    cout << "}\n";
}
```

- Meccanismo efficiente della reference
- Accesso in sola lettura
 - Verificato dal compilatore

non spreca tempo



Pass-by-value vs pass-by-reference

```
void g(int a, int& r, const int& cr)
{
    ++a;
    ++r;
    int x = cr;
}
```

Una reference si attacca solo a oggetti scrivibili.
Una constant reference si può attaccare anche a un'oggetto non scrivibile, perché tanto non può modificarne il valore (il compilatore crea comunque una variabile, ma è nascosta)

```
int main()
{
    int x = 0;
    int y = 0;
    int z = 0;

    g(x, y, z);           // x == 0; y == 1; z == 0;
    g(1, 2, 3);           // errore: ref. richiede un oggetto
    g(1, y, 3);           // OK: const ref. accetta un literal
}
```

The diagram illustrates the binding of references in the `g` function. For the first call `g(x, y, z)`, all three arguments are regular variables. For the second call `g(1, 2, 3)`, the first two are literals, and the third is a reference to the variable `z`. For the third call `g(1, y, 3)`, the first two are literals, and the third is a reference to the literal value `3`.



Caratteristiche del passaggio per ref.

- Una reference richiede un lvalue
 - Possiamo scriverci!
- Una const reference non richiede un lvalue
 - L'ultima chiamata diventa:

```
g(1, y, 3); // ->
```

```
// int __compiler_generated = 3;  
// g(1, y, __compiler_generated);
```

```
void g(int a, int& r, const int& cr)  
{  
    // ...  
}
```

- Temporary (object)



Copia vs reference

- Ora conosciamo due modi per passare un argomento...
 - Quale scegliere?



Copia vs reference

- Passaggio per valore per oggetti piccoli
- Passaggio per const-reference per grandi oggetti da non modificare
- Preferibile ritornare un risultato rispetto a modificare un oggetto usando una reference
- Usare una reference se è proprio necessario
 - Manipolare contenitori (es, vector) e altri oggetti grandi
 - Funzioni che devono produrre più di un output
 - Si suppone che una funzione che accetta una ref modifichi l'oggetto passato

Conversione degli argomenti



Check degli argomenti

- Un argomento passato a una funzione è convertito nel tipo del parametro

```
void f(T x);
f(y);
T x = y;           // inizializza x con y

// f(y); è legale tutte le volte che T x = y; lo è
// entrambe le x hanno lo stesso valore
```

```
void f(double x);

void g(int y)
{
    f(y);                // conversione
    double x = y;        // inizializza x con y (conversione)
}
```



Conversioni automatiche

- Utili, ma possono produrre risultati indesiderati

```
void ff(int x);

void gg(double y)
{
    ff(y);           // come posso sapere se è sensato?
    int x = y;       // come posso sapere se è sensato?
}
```

- Difficile dire se questo codice ha un errore o no



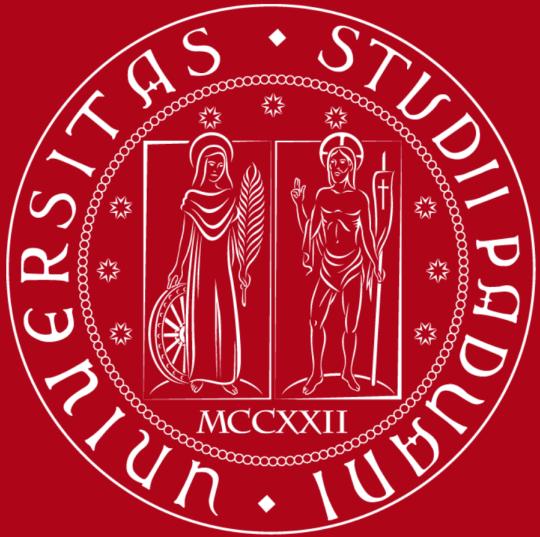
Conversioni esplicite

- Un cast esplicito è più espressivo: un altro programmatore capisce che è intenzionale

```
void ggg(double x)
{
    int x1 = x;                                // troncamento
    int x2 = int(x);
    int x3 = static_cast<int> (x);   // conversione esplicita

    ff(x1);
    ff(x2);
    ff(x3);

    ff(x);
    ff(int(x));
    ff(static_cast<int> (x));           // conversione esplicita
}
```



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Reference

Stefano Ghidoni