3.3 – Funzioni e reference, efficienza

Libro di testo:

- Capitolo 4
- Capitolo 8.5

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Funzioni
- Passaggio di argomenti per copia e per riferimento
- Questioni di efficienza

Funzioni – recap

- Suddividono la computazione in blocchi logici e funzionali
- Input: argomenti (0+)
- Output: valore di ritorno (0 o 1)
- Passare un argomento significa inizializzare un argomento formale con il valore dell'argomento

```
double fct(int a, double d);
double fct(int a, double d) { return a*d; }
```

Parametri / argomenti formali (formal arguments)

```
// ...
double d2 = fct(i, d1);
```

Argomenti

Passaggio per valore

- Passaggio di argomenti per valore
- Un argomento è una variabile locale della funzione
  - Inizializzata a ogni chiamata
  - **Valore copiato**

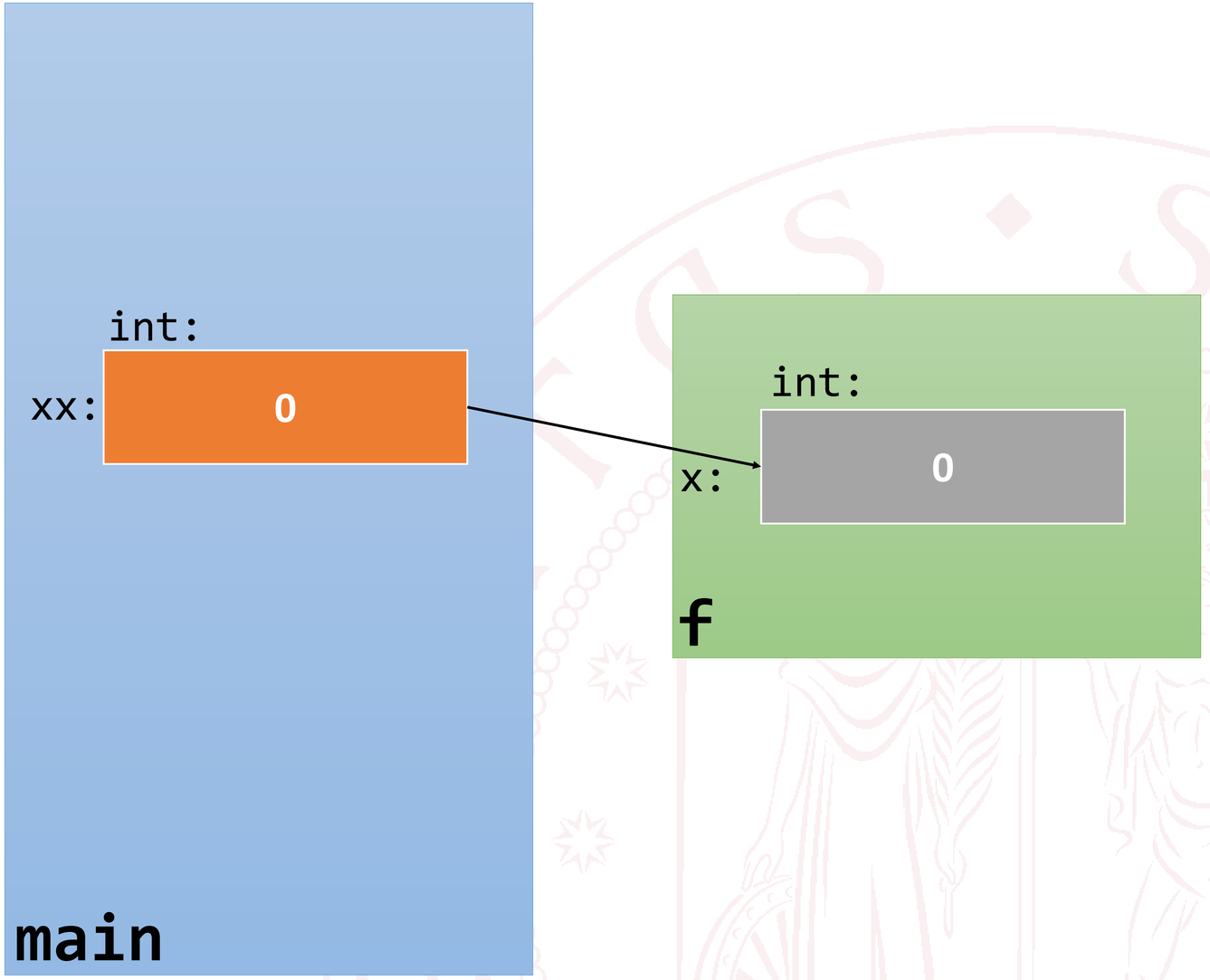
Passaggio per valore

```
int f(int x)
{
    x = x + 1;
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n';
    cout << xx << '\n';

    int yy = 7;
    cout << f(yy) << '\n';
    cout << yy << '\n';

    return 0;
}
```



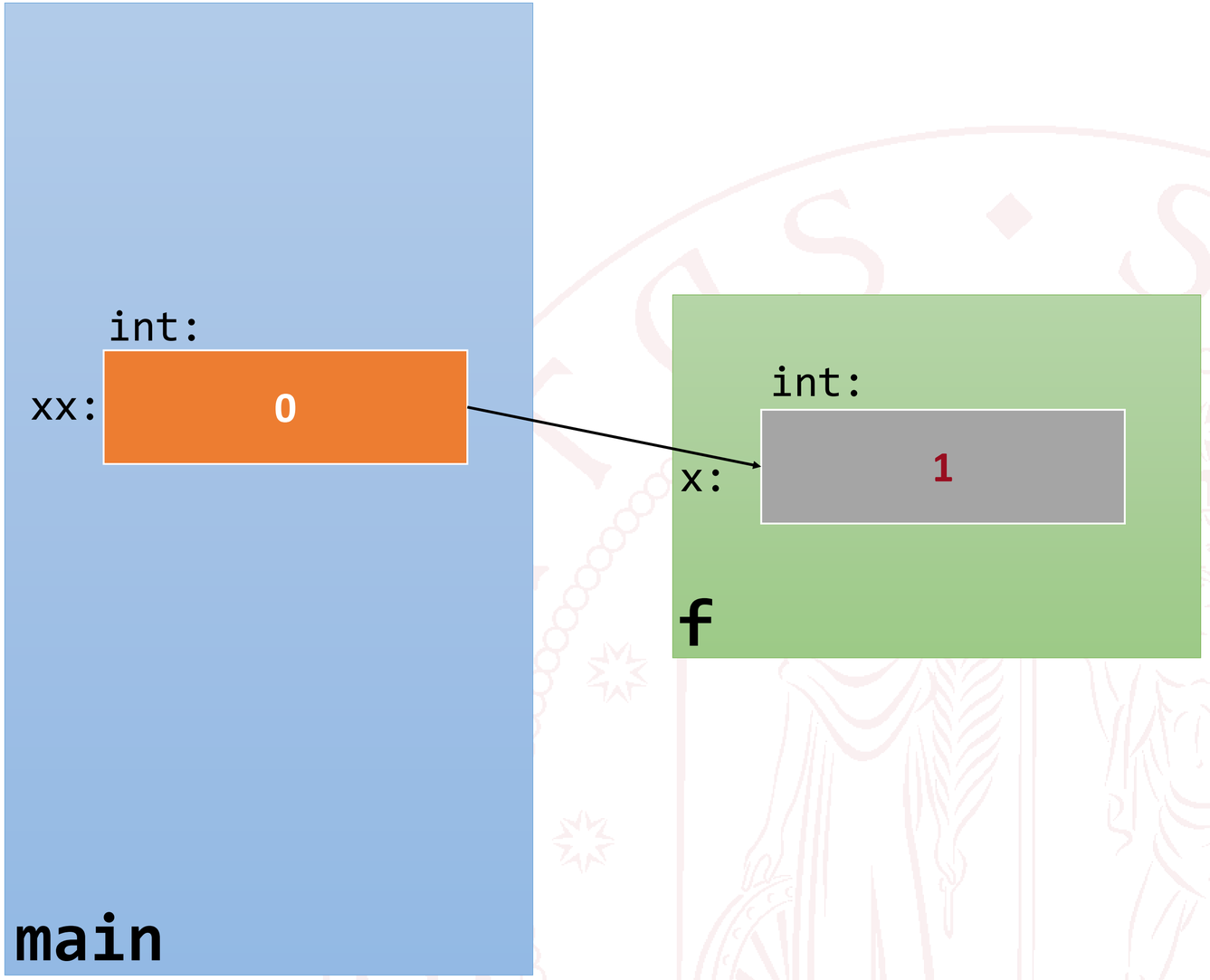
Passaggio per valore

```
int f(int x)
{
    x = x + 1;
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n';
    cout << xx << '\n';

    int yy = 7;
    cout << f(yy) << '\n';
    cout << yy << '\n';

    return 0;
}
```



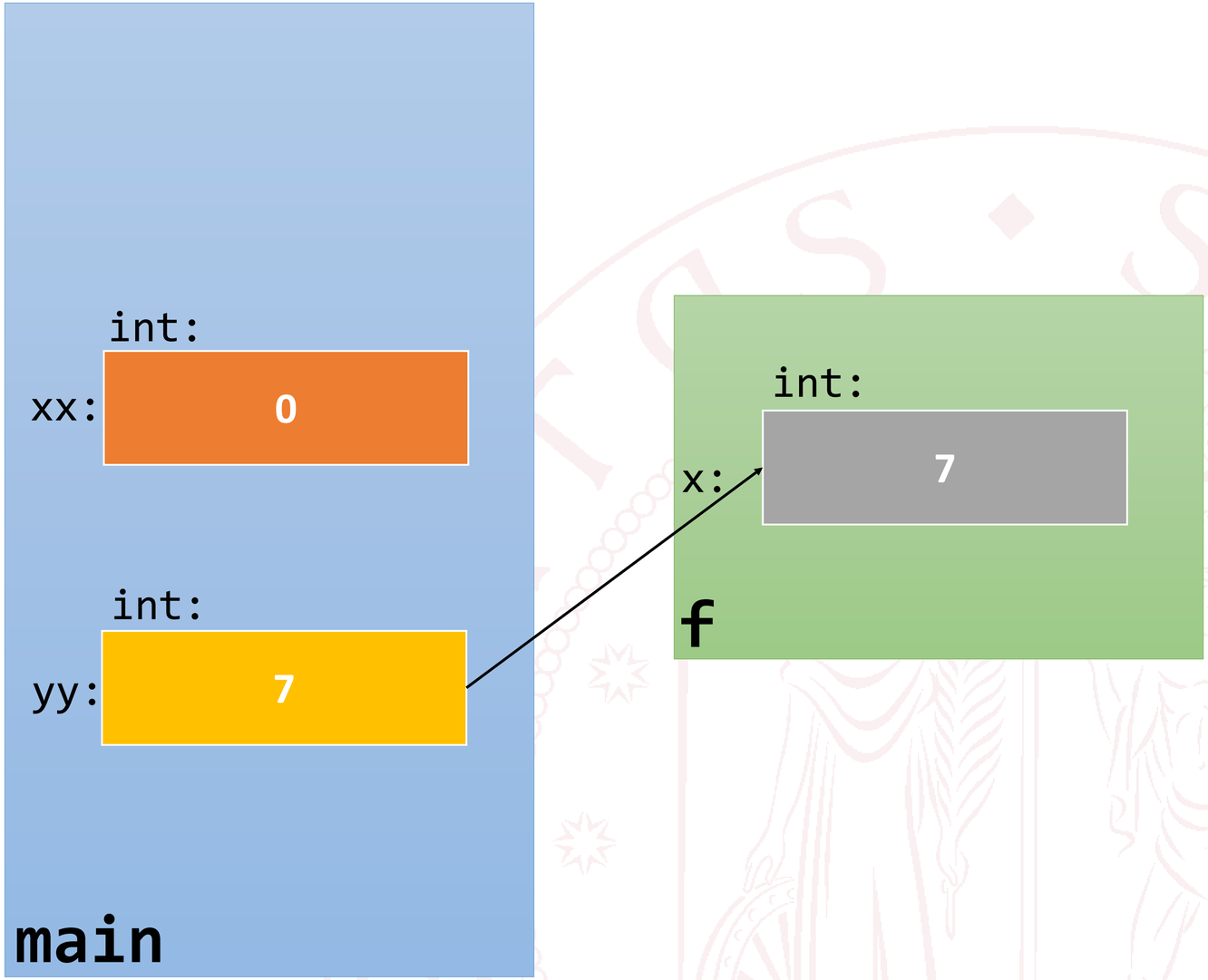
Passaggio per valore

```
int f(int x)
{
    x = x + 1;
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n';
    cout << xx << '\n';

    int yy = 7;
    cout << f(yy) << '\n';
    cout << yy << '\n';

    return 0;
}
```



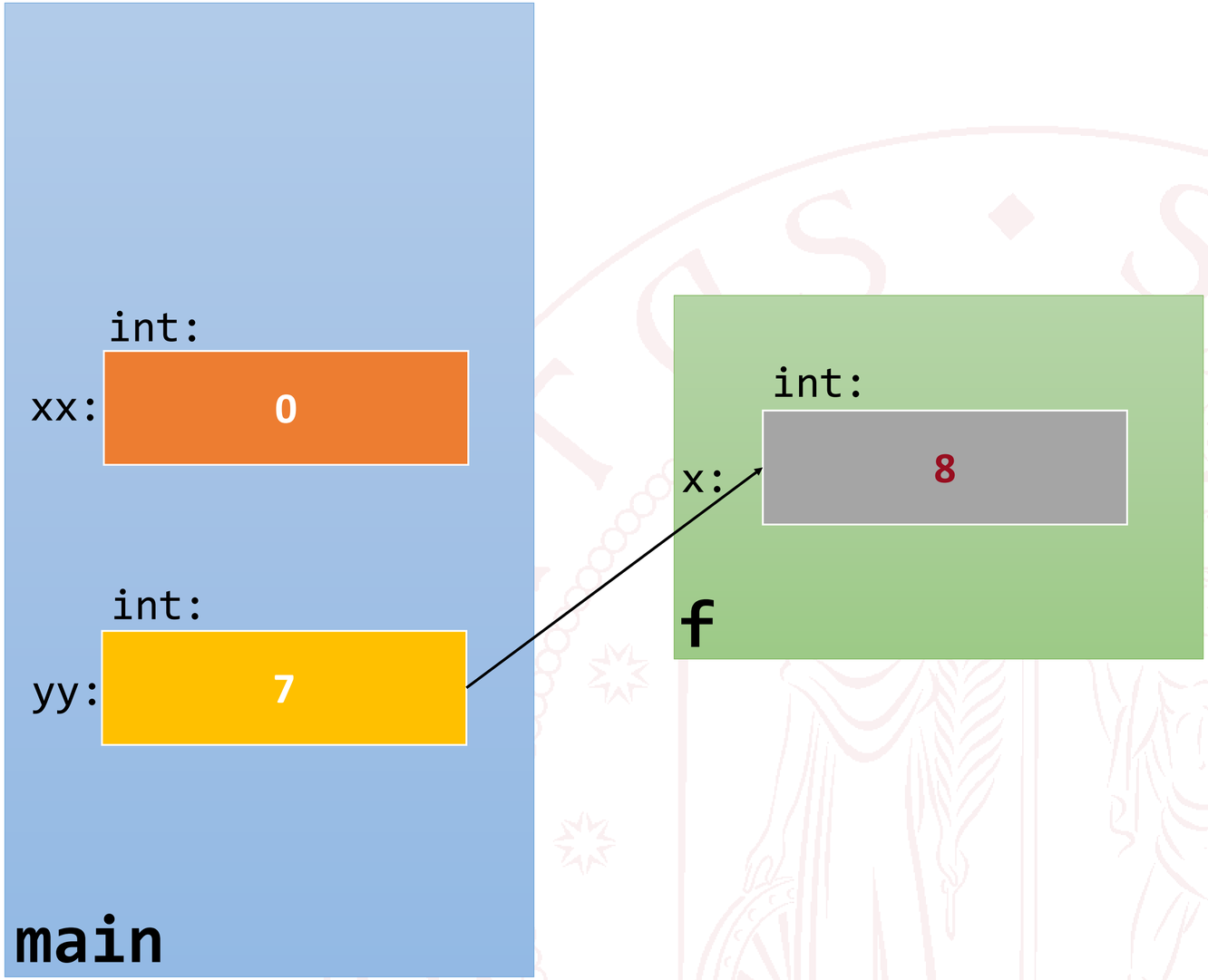
Passaggio per valore

```
int f(int x)
{
    x = x + 1;
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n';
    cout << xx << '\n';

    int yy = 7;
    cout << f(yy) << '\n';
    cout << yy << '\n';

    return 0;
}
```



Passaggio per valore

```
int f(int x)
{
    x = x + 1;
    return x;
}

int main()
{
    int xx = 0;
    std::cout << f(xx) << '\n';
    std::cout << xx << '\n';

    int yy = 7;
    std::cout << f(yy) << '\n';
    std::cout << yy << '\n';

    return 0;
}
```

← Qual è l'output?
← Qual è l'output?

← Qual è l'output?
← Qual è l'output?

Passaggio per valore

- Molto comodo, efficace ed efficiente
  - Pochi dati (es., tipi standard)
  - Dati che non devono essere modificati
- Potenziali problemi di efficienza
  - Molti dati (la copia è dispendiosa)
    - Vettori
    - Immagini, db, testi, ...

Problemi di efficienza

`std::vector` (excursus)

- `std::vector`: è la versione moderna dell'array C
  - Gestisce vettori di ogni tipo di dato
    - `std::vector<int> vi; std::vector<double> vd; std::vector<T> vt;`
    - `<>` contiene un tipo!
  - Conosce quanti elementi ha in memoria
    - `vi.size()`
  - Gestisce automaticamente la memoria
  - Funzioni per aggiungere elementi
    - `v.push_back()`
  - Costruttore con inizializzazione
    - `std::vector<int> vi(10);`
- Ulteriori info: http://www.cplusplus.com/reference/vector/vector/

Problemi di efficienza

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

- **Cosa fa questa funzione?**

Problemi di efficienza

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

Potenziale problema di efficienza

- **È davvero un problema?**

Problemi di efficienza

- È davvero un problema?

```
void int f(int x)
{
    std::vector<double> vd1 (10);
    std::vector<double> vd2(1000000);
    std::vector<double> vd3(x)
    // ... riempimento di vd1, vd2, vd3 ...
    print(vd1);
    print(vd2);
    print(vd3);
}
```

In quali di questi casi?

- Non troppa ossessione per l'efficienza, ma attenzione ai dettagli

Migliorare l'efficienza

- Evitare la copia
  - Accedere direttamente ai dati passati come argomento
- Quale strumento usare?

Migliorare l'efficienza

- Evitare la copia
  - Accedere direttamente ai dati passati come argomento
- Quale strumento usare?
- Indirizzo del dato
  - Puntatore?
  - Riferimento (reference)

Reference

Passaggio per riferimento

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

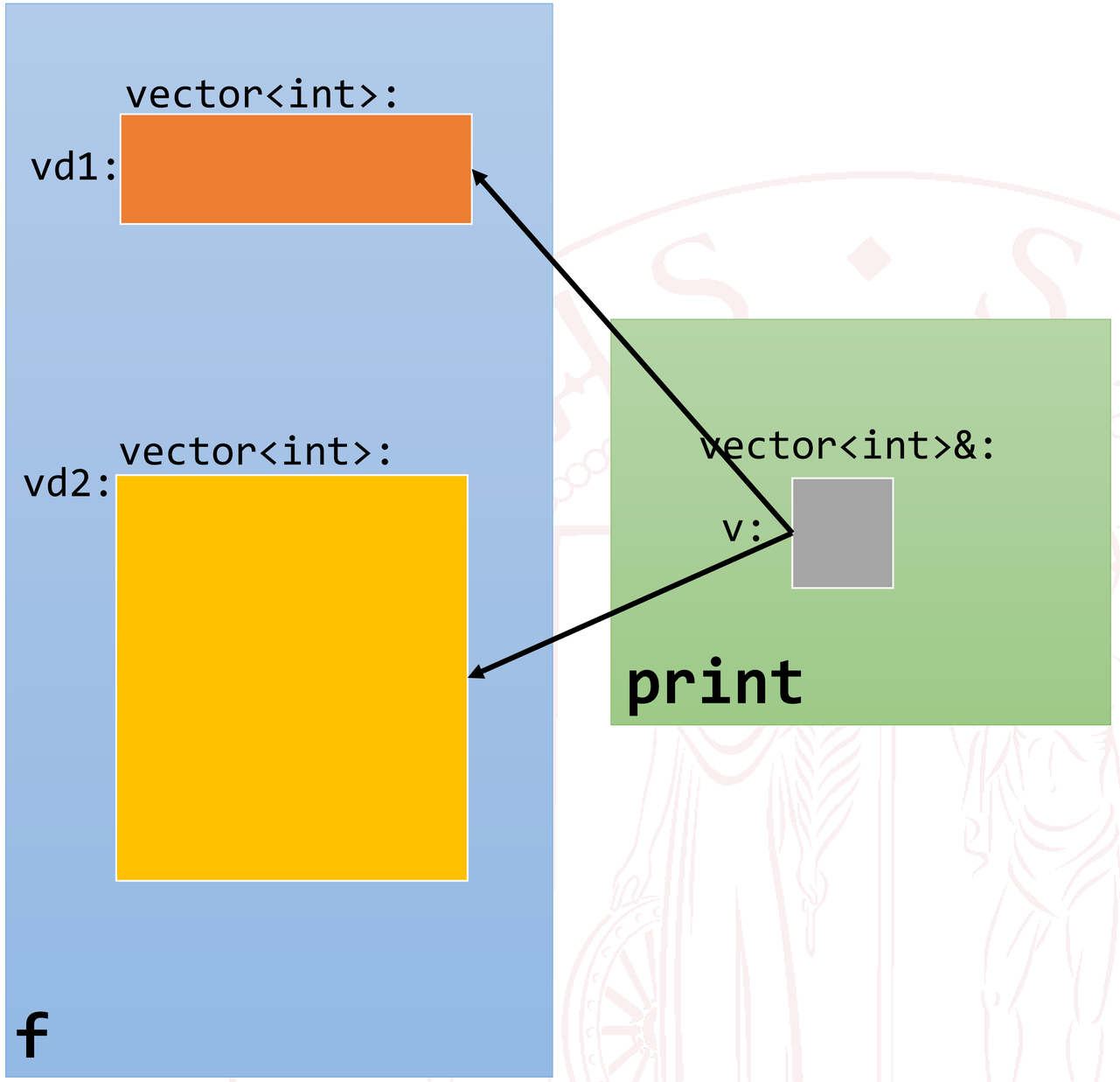
```
void int f(int x)
{
    std::vector<double> vd1 (10);
    std::vector<double> vd2(1000000);
    std::vector<double> vd3(x)
    // ... riempimento di vd1, vd2, vd3 ...
    print(vd1);
    print(vd2);
    print(vd3);
}
```

- **Trovate le differenze!**
- Una piccolissima differenza di sintassi
  - Ora l'argomento di `print` *si riferisce* (è un riferimento!) all'oggetto definito nel chiamante
- La funzione `f` è identica al precedente
  - **È un problema?**

Passaggio per riferimento

```
void int f(int x)
{
    vector<double> vd1 (10);
    vector<double> vd2(1000000);
    vector<double> vd3(x)
    //... riempimento di vd1, vd2, vd3...
    print(vd1);
    print(vd2);
    print(vd3);
}
```

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



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

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

- **i e r sono alias!**
  - Anche in scrittura

Reference

- Considerate questo esempio:

```
std::vector< std::vector<double> > v;


double val = v[f(x)][g(y)];          // comodo se dobbiamo accedere in
                                     // lettura a quel valore molte volte

// 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?
- L'accesso in sola lettura è molto importante
- 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

Pass-by-value vs pass-by-reference

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

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

    g(x, y, z);    // ?
    g(1, 2, 3);    // ?
    g(1, y, 3);    // ?
}
```

Pass-by-value vs pass-by-reference

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

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);    // ?
    g(1, y, 3);    // ?
}
```

Pass-by-value vs pass-by-reference

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

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);    // ?
}
```

Pass-by-value vs pass-by-reference

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

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
}
```

Caratteristiche del passaggio per reference

- 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);
```

  - Temporary (object)

Copia vs reference

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

Riepilogo

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

Recap

- Elementi tecnici delle funzioni
- Gestione degli argomenti
  - Per copia vs per riferimento
- Reference e const reference
  - Cenni agli effetti sulla memoria
- Conversioni automatiche degli argomenti