

# Fondamenti di Programmazione (A)

## I0 - Regole di scoping

Vincenzo Arceri - Università degli Studi di Parma - [vincenzo.arceri@unipr.it](mailto:vincenzo.arceri@unipr.it)

# Puntate precedenti

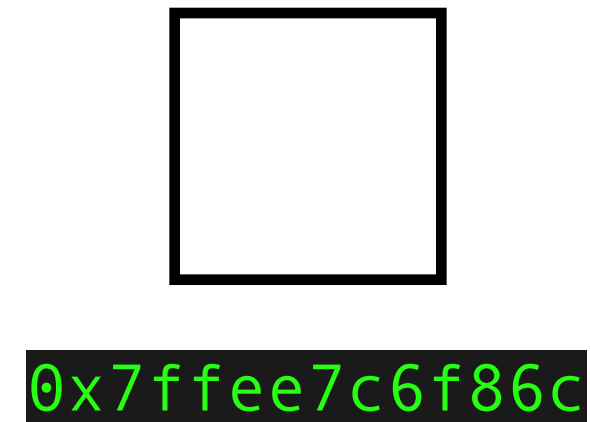
- Assegnamenti e dichiarazioni
- Blocchi
- Statement `if` (comando di selezione)
- Statement iterativi
  - ◆ `while`
  - ◆ `do – while`
  - ◆ `for`

# Dichiarazione di variabile in C++

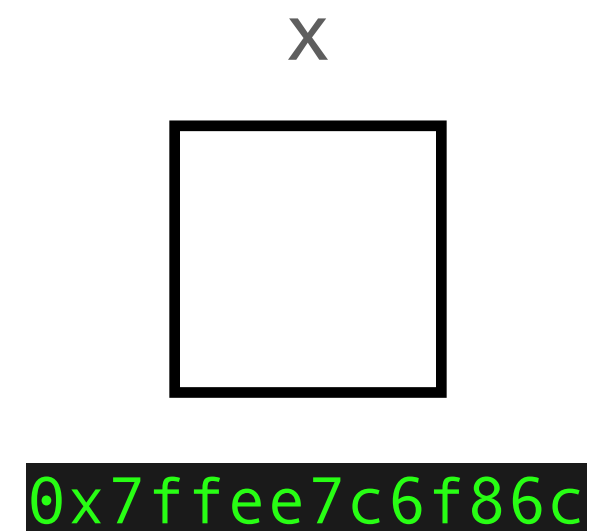
Dalla Lezione 7

```
int x;
```

- Creazione della variabile (allocazione di memoria)



- Associazione/*binding* fra la variabile e il suo nome



# Ambiente

- Insieme di associazioni fra nomi (identificatori) ed *oggetti denotabili* in un punto preciso di un programma e in un momento preciso dell'esecuzione

# Ambiente

- Insieme di associazioni fra nomi (identificatori) ed *oggetti denotabili* in un punto preciso di un programma e in un momento preciso dell'esecuzione
- **Oggetti denotabili:** oggetti a cui può essere dato
  - Variabili, tipi, funzioni
  - ...
- Una dichiarazione di variabile aggiunge una nuova associazione nome-oggetto denotabile all'ambiente

# Ambiente

- Insieme di associazioni fra nomi (identificatori) ed *oggetti denotabili* in un punto preciso di un programma e in un momento preciso dell'esecuzione
- **Oggetti denotabili:** oggetti a cui può essere dato
  - Variabili, tipi, funzioni
  - ...
- Una dichiarazione di variabile aggiunge una nuova associazione nome-oggetto denotabile all'ambiente

Per quanto tempo è valida quell'associazione?

# Scoping

- Il **campo d'azione (scope)** di una dichiarazione è l'insieme di parti di programma in cui il binding nome-oggetto denotabile è **visibile**

# Scoping

- Il **campo d'azione (scope)** di una dichiarazione è l'insieme di parti di programma in cui il binding nome-oggetto denotabile è **visibile**
- Regole di scope: sono le regole che determinano la visibilità di un'associazione nome-oggetto denotabile



# Blocchi e scoping

- Il blocco è una regione testuale del programma utilizzata per raggruppare più comandi

```
{  
    x = 1;  
    y = 3;  
    int z = 1;  
    z = z * x + y;  
}
```

# Blocchi e scoping

- Il blocco è una regione testuale del programma utilizzata per raggruppare più comandi

```
{  
    x = 1;  
    y = 3;  
    int z = 1;  
    z = z * x + y;  
}
```

- **Regola di scope**

Ogni dichiarazione contenuta in un blocco B si dice **locale (al blocco B)** ed è valida dal punto in cui compare fino alla fine del blocco

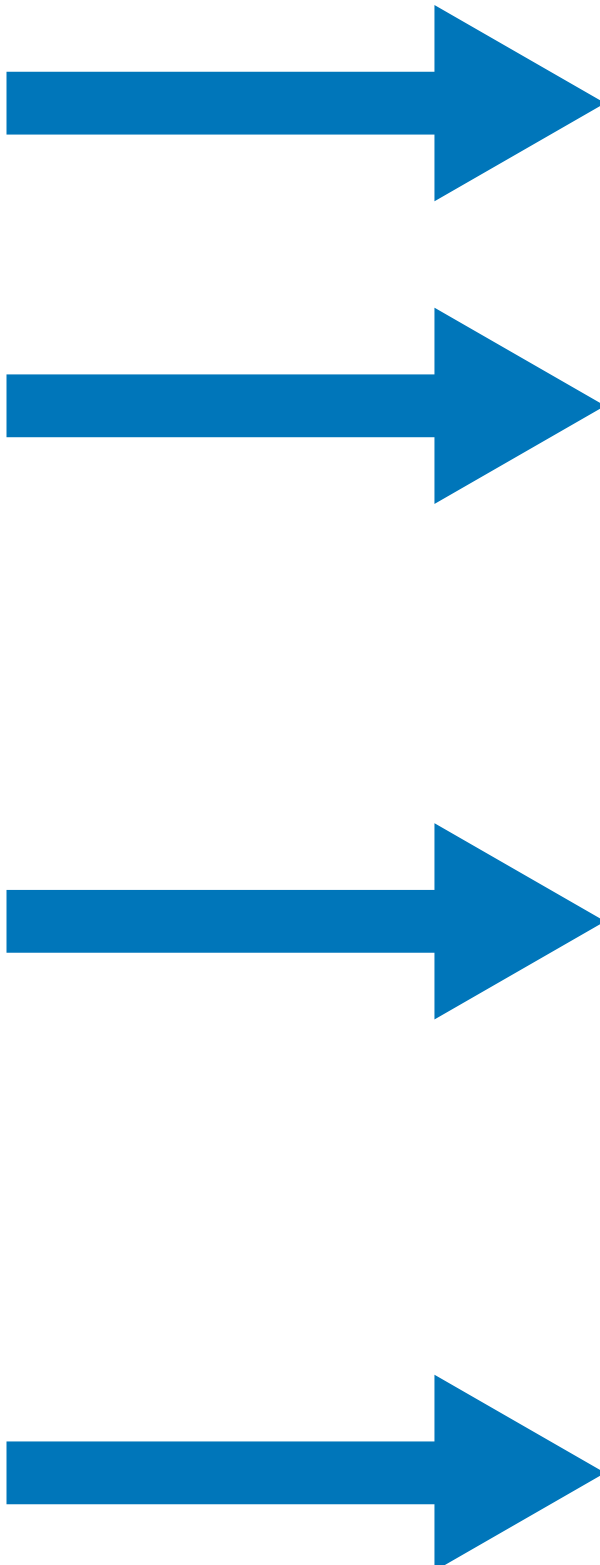
# Blocchi e scoping

## Esempio

```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
    }  
    {  
        int y = 3;  
        y++;  
    }  
    x++;  
    int w = x;  
}
```

# Blocchi e scoping

## Esempio



```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
    }  
    {  
        int y = 3;  
        y++;  
    }  
    x++;  
    int w = x;  
}
```

# Blocchi e scoping

## Esempio

```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
    }  
    {  
        int y = 3;  
        y++;  
    }  
    x++;  
    int w = x;  
}
```

# Blocchi e scoping

## Esempio

```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
    }
    {
        int y = 3;
        y++;
    }
    x++;
    int w = x;
}
```

Scope di int x = 1

# Blocchi e scoping

## Esempio

```
{  
  int x = 1;  
  {  
    int z = 1;  
    z = z * x + 1;           Scope di int z = 1  
  }  
  {  
    int y = 3;  
    y++;  
  }  
  x++;  
  int w = x;                Scope di int x = 1  
}
```

# Blocchi e scoping

## Esempio

```
{  
  int x = 1;  
  {  
    int z = 1;  
    z = z * x + 1;           Scope di int z = 1  
  }  
  {  
    int y = 3;  
    y++;                     Scope di int y = 3  
  }  
  x++;  
  int w = x;                 Scope di int x = 1  
}
```



# Blocchi e scoping

## Esempio

```
{  
  int x = 1;  
  {  
    int z = 1;  
    z = z * x + 1;           Scope di int z = 1  
  }  
  {  
    int y = 3;  
    y++;                     Scope di int y = 3  
  }  
  x++;  
  int w = x;                 Scope di int w = x  
                             Scope di int x = 1  
}
```

# Blocchi e scoping

## Sotto-blocchi

- **Regola di scope**

Ogni dichiarazione contenuta in un blocco B si dice **locale (al blocco B)** ed è valida dal punto in cui compare fino alla fine del blocco

Ogni dichiarazione contenuta in un blocco B si estende anche ai suoi sotto-blocchi **a meno che** non intervenga nel sotto-blocco una dichiarazione con lo stesso nome

# Blocchi e scoping

Esempio



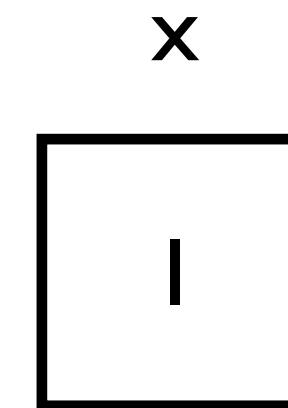
```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
        {  
            int x = 5;  
            z = x * 2;  
        }  
        x++;  
        cout << x;  
    }  
}
```

# Blocchi e scoping

## Esempio



```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
        {  
            int x = 5;  
            z = x * 2;  
        }  
        x++;  
        cout << x;  
    }  
}
```

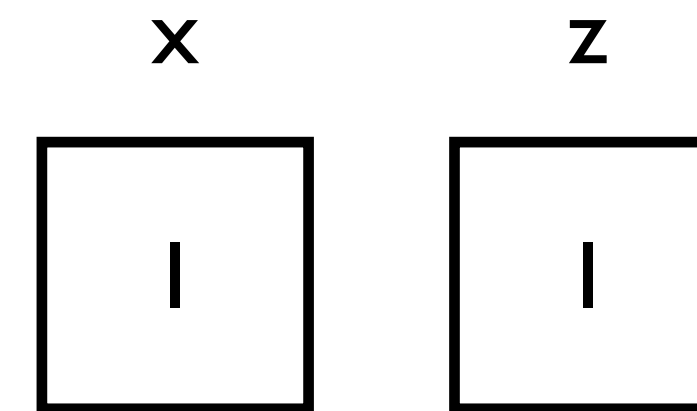


# Blocchi e scoping

## Esempio



```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
        {
            int x = 5;
            z = x * 2;
        }
        x++;
        cout << x;
    }
}
```

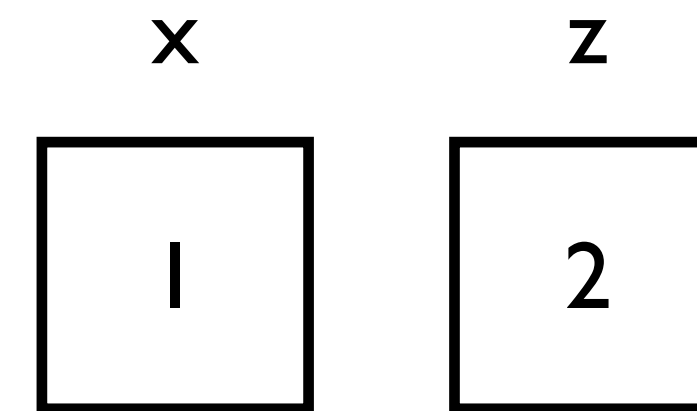


# Blocchi e scoping

## Esempio



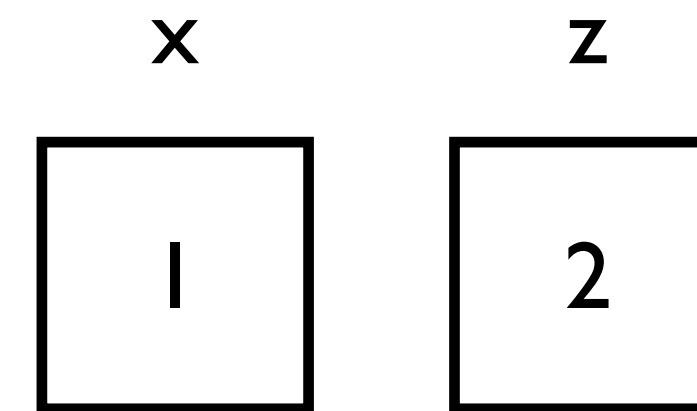
```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
        {  
            int x = 5;  
            z = x * 2;  
        }  
        x++;  
        cout << x;  
    }  
}
```



# Blocchi e scoping

## Esempio

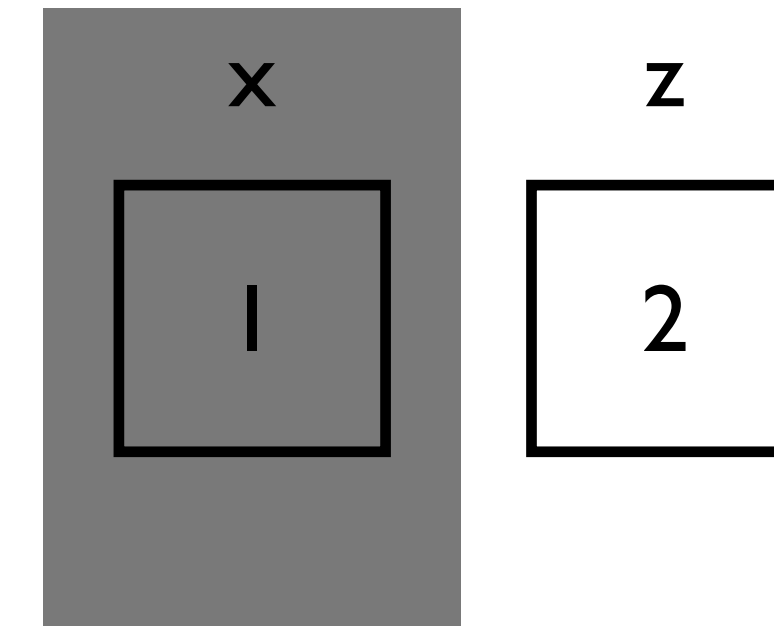
```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
        {  
            int x = 5;  
            z = x * 2;  
        }  
        x++;  
        cout << x;  
    }  
}
```



# Blocchi e scoping

## Esempio

```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
        {
            int x = 5;
            z = x * 2;
        }
        x++;
        cout << x;
    }
}
```



### Shadowing

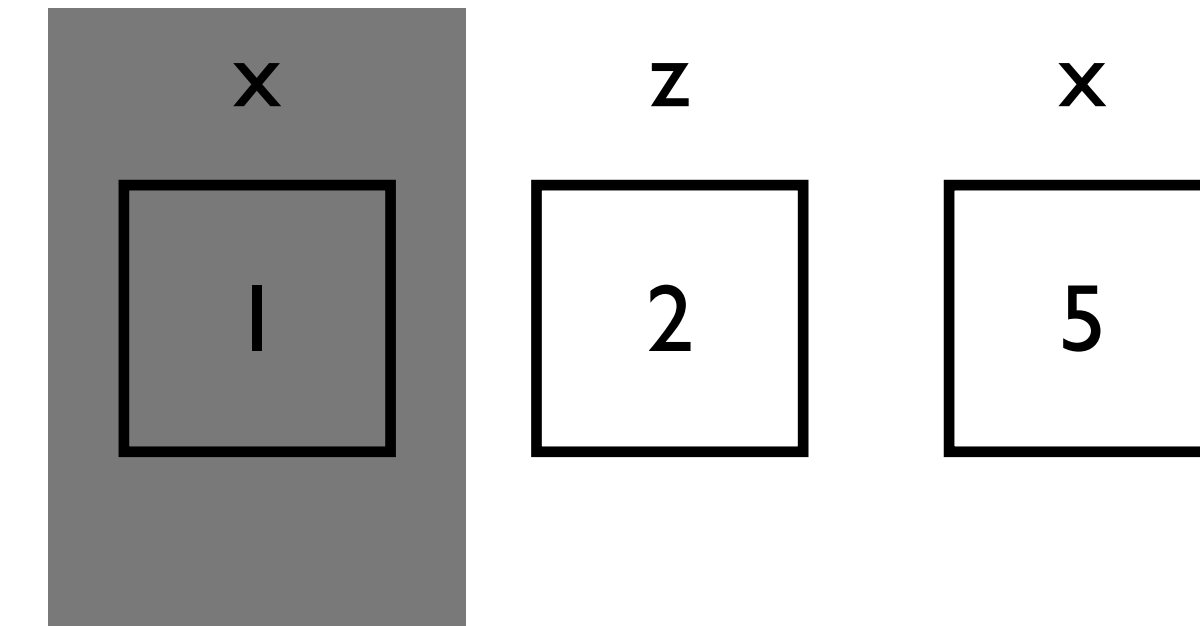
La dichiarazione viene oscurata:  
l'associazione per il nome creato  
viene **disattivata** per tutto il  
blocco interno e verrà **riattivata**  
all'uscita al blocco



# Blocchi e scoping

## Esempio

```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
        {
            int x = 5;
            z = x * 2;
        }
        x++;
        cout << x;
    }
}
```



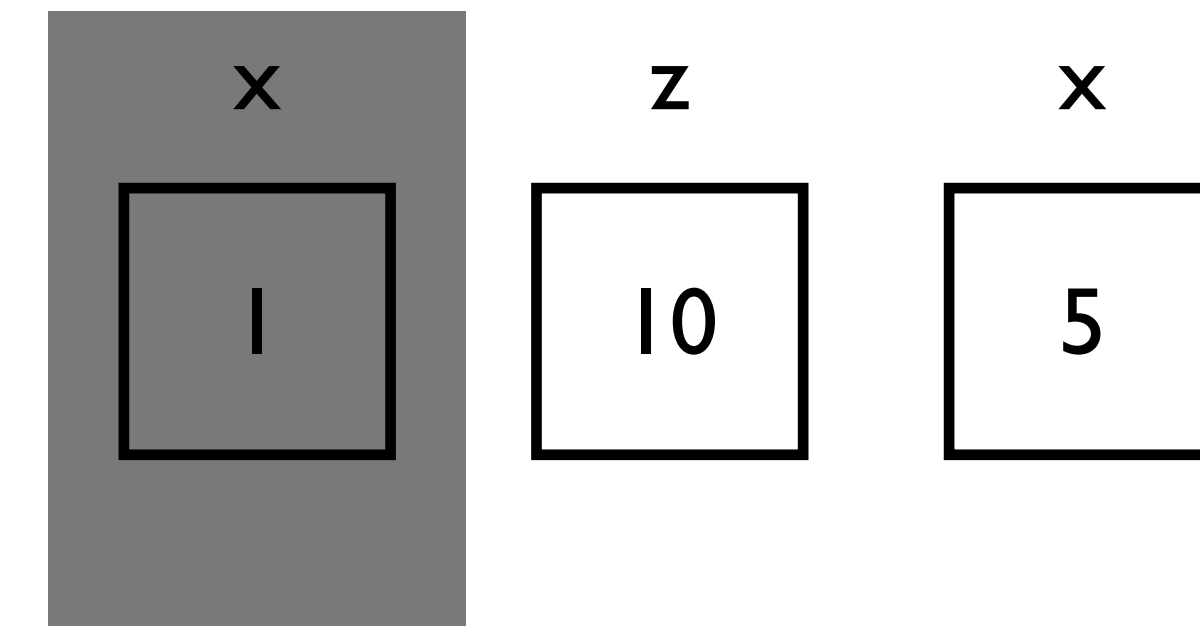
### Shadowing

La dichiarazione viene oscurata:  
l'associazione per il nome creato  
viene **disattivata** per tutto il  
blocco interno e verrà **riattivata**  
all'uscita al blocco

# Blocchi e scoping

## Esempio

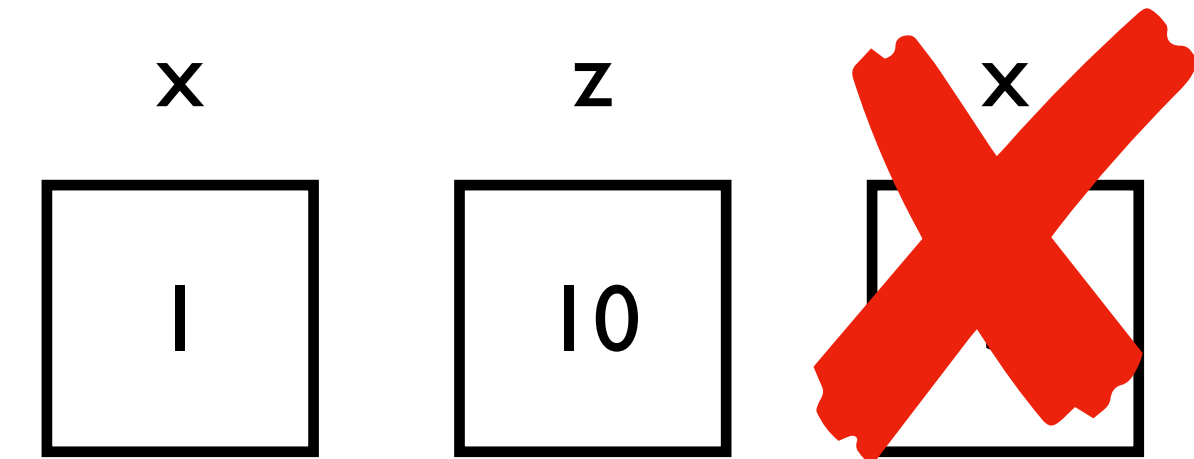
```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
        {
            int x = 5;
            z = x * 2;
        }
        x++;
        cout << x;
    }
}
```



# Blocchi e scoping

## Esempio

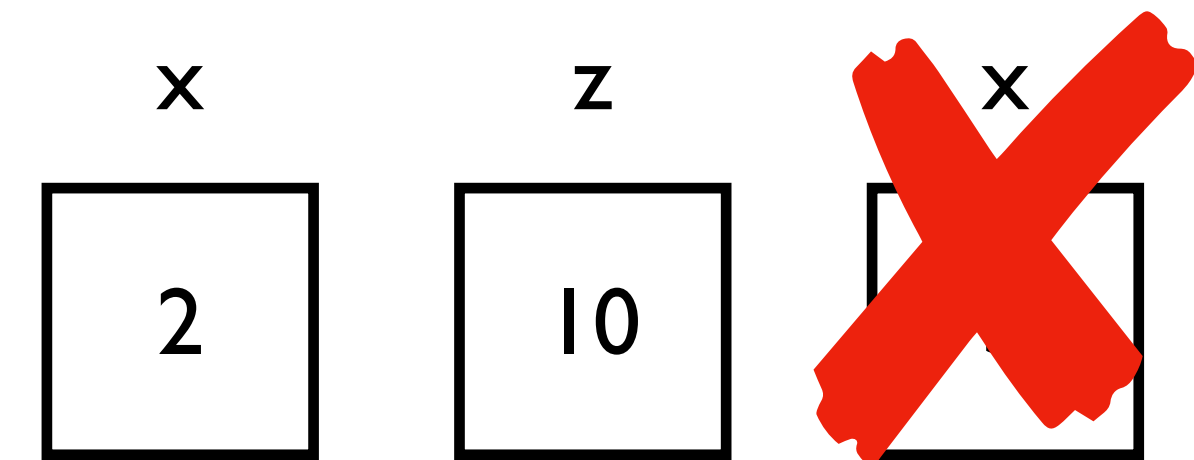
```
{
    int x = 1;
    {
        int z = 1;
        z = z * x + 1;
        {
            int x = 5;
            z = x * 2;
        }
        x++;
        cout << x;
    }
}
```



# Blocchi e scoping

## Esempio

```
{  
    int x = 1;  
    {  
        int z = 1;  
        z = z * x + 1;  
        {  
            int x = 5;  
            z = x * 2;  
        }  
        x++;  
        cout << x;  
    }  
}
```



# Tipi di associazioni

```
const int max = 100;  
int n = max;
```

```
int main() {  
    int x = 0;  
    int y = 1;  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```

Dichiarazioni **locali** al blocco del main  
(poiché visibile nel blocco del main **e**  
interni al blocco del main)

Dichiarazioni **non locali** al blocco  
del for (poiché visibile nel blocco  
del for **ma** esterna al blocco del  
for)

# Tipi di associazioni

```
const int max = 100;  
int n = max;
```

```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```

Dichiarazioni **locali** al  
blocco del for



# Tipi di associazioni

```
const int max = 100;  
int n = max;
```


Dichiarazioni **non locali**  
per nessun blocco

```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```

Dichiarazioni **globali**:  
scritte fuori da ogni  
blocco del programma e  
quindi visibili in tutto il  
programma

# Operazioni sull'ambiente

```
const int max = 100;  
int n = max;
```



```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```



**Creazione dell'associazione nome-  
oggetto denotabile**



# Operazioni sull'ambiente

```
const int max = 100;  
int n = max;
```

```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```



**Disattivazione di un'associazione nome-  
oggetto denotabile**

# Operazioni sull'ambiente

```
const int max = 100;  
int n = max;
```

```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```



**Riattivazione di un'associazione nome-  
oggetto denotabile**

# Operazioni sull'ambiente

```
const int max = 100;  
int n = max;
```

```
int main() {  
    int x = 0;  
    int y = 1  
    cout << n;  
    cout << x;  
    for (int i = 0; i < n; i++) {  
        int x = 1;  
        cout << x << endl;  
        x = y + 1;  
    }  
    cout << x;  
    return 0;  
}
```



**Distruzione di un'associazione nome-  
oggetto denotabile**

# Statement switch

# Statement switch

- **Problema:** preso in input un intero positivo  $1 \leq n \leq 12$ , stampare a video il mese corrispondente

# Statement `switch`

- Nel caso in cui è necessario effettuare un insieme di operazioni in base ai  $k$  valori possibili di un'espressione *exp*, C++ offre un costrutto più naturale rispetto a una serie di `if – else`

# Statement switch

- **Problema:** preso in input un intero positivo  $1 \leq n \leq 12$ , stampare a video il mese corrispondente

# Statement switch

```
switch (exp) {  
    case  $c_0$  :  
        seqstmt0  
    case  $c_1$  :  
        seqstmt1  
    case  $c_2$  :  
        seqstmt2  
  
    ...  
  
    case  $c_n$  :  
        seqstmt $n$   
    default :  
        seqstmt $n+1$   
}
```

- *exp* è un'espressione compatibile con `int`
- $\forall i \in [0, n]$   $c_i$  è un'espressione costante
- *seqstmt* sono sequenze di comandi (non necessariamente **un** comando come nel caso `if` o `while`)



# Statement switch

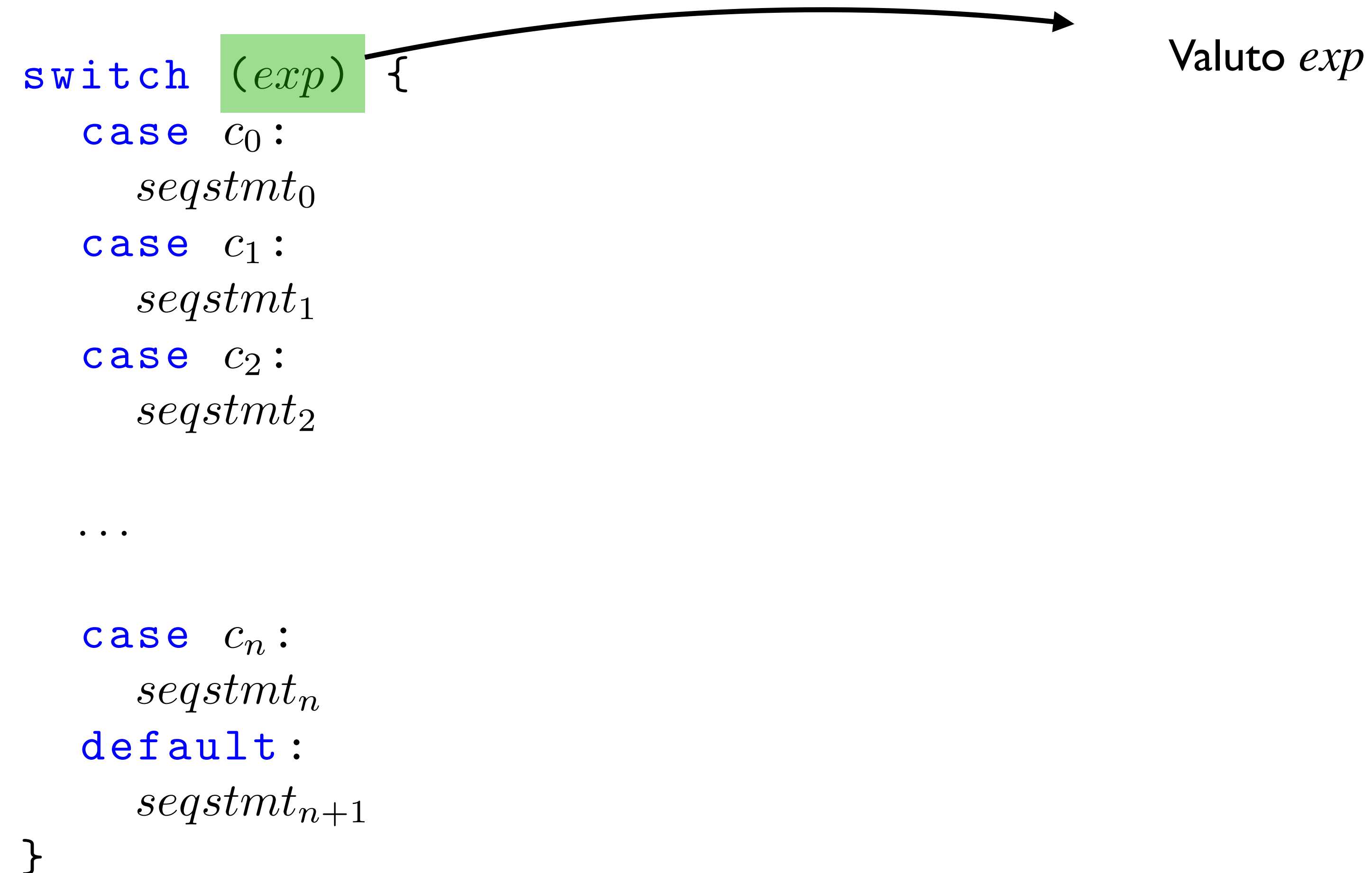
```
switch (exp) {  
    case  $c_0$  :  
        seqstmt0  
    case  $c_1$  :  
        seqstmt1  
    case  $c_2$  :  
        seqstmt2  
  
    ...  
  
    case  $c_n$  :  
        seqstmt $n$   
    default :  
        seqstmt $n+1$   
}
```

- *exp* è un'espressione compatibile con `int`
- $\forall i \in [0, n]$   $c_i$  è un'espressione costante
- *seqstmt* sono sequenze di comandi (non necessariamente **un** comando come nel caso `if` o `while`)

Il caso default è opzionale!

# Statement switch

## Semantica informale



# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case c0 :  
    seqstmt0
```

```
  case c1 :  
    seqstmt1
```

```
  case c2 :  
    seqstmt2
```

```
  ...
```

```
  case cn :  
    seqstmtn
```

```
  default :  
    seqstmtn+1
```

```
}
```

$exp == c_0?$

Supponiamo false

# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case  $c_0$ :
```

```
     $seqstmt_0$ 
```

```
  case  $c_1$ :
```

```
     $seqstmt_1$ 
```

```
  case  $c_2$ :
```

```
     $seqstmt_2$ 
```

```
  ...
```

```
  case  $c_n$ :
```

```
     $seqstmt_n$ 
```

```
  default:
```

```
     $seqstmt_{n+1}$ 
```

```
}
```

$exp == c_1?$

Supponiamo true

# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case  $c_0$ :
```

```
    seqstmt0
```

```
  case  $c_1$ :
```

```
    seqstmt1
```

```
  case  $c_2$ :
```

```
    seqstmt2
```

```
  ...
```

```
  case  $c_n$ :
```

```
    seqstmt $n$ 
```

```
  default:
```

```
    seqstmt $n+1$ 
```

```
}
```

esegue *seqstmt*<sub>1</sub>

# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case  $c_0$ :
```

```
    seqstmt0
```

```
  case  $c_1$ :
```

```
    seqstmt1
```

```
  case  $c_2$ :
```

```
    seqstmt2
```

```
  ...
```

```
  case  $c_n$ :
```

```
    seqstmt $n$ 
```

```
  default:
```

```
    seqstmt $n+1$ 
```

```
}
```

esegue *seqstmt*<sub>2</sub>

Senza controllare se  $exp == c_2$ !

# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case  $c_0$ :
```

```
    seqstmt0
```

```
  case  $c_1$ :
```

```
    seqstmt1
```

```
  case  $c_2$ :
```

```
    seqstmt2
```

```
  ...
```

```
  case  $c_n$ :
```

```
    seqstmtn
```

```
  default:
```

```
    seqstmtn+1
```

```
}
```

esegue *seqstmt*<sub>*n*</sub> Senza controllare se *exp* == *c*<sub>*n*</sub>!

# Statement switch

## Semantica informale

```
switch (exp) {
```

```
  case  $c_0$ :
```

```
     $seqstmt_0$ 
```

```
  case  $c_1$ :
```

```
     $seqstmt_1$ 
```

```
  case  $c_2$ :
```

```
     $seqstmt_2$ 
```

```
  ...
```

```
  case  $c_n$ :
```

```
     $seqstmt_n$ 
```

```
  default:
```

```
     $seqstmt_{n+1}$ 
```

```
}
```

esegue  $seqstmt_{n+1}$





# Statement switch

## Semantica informale

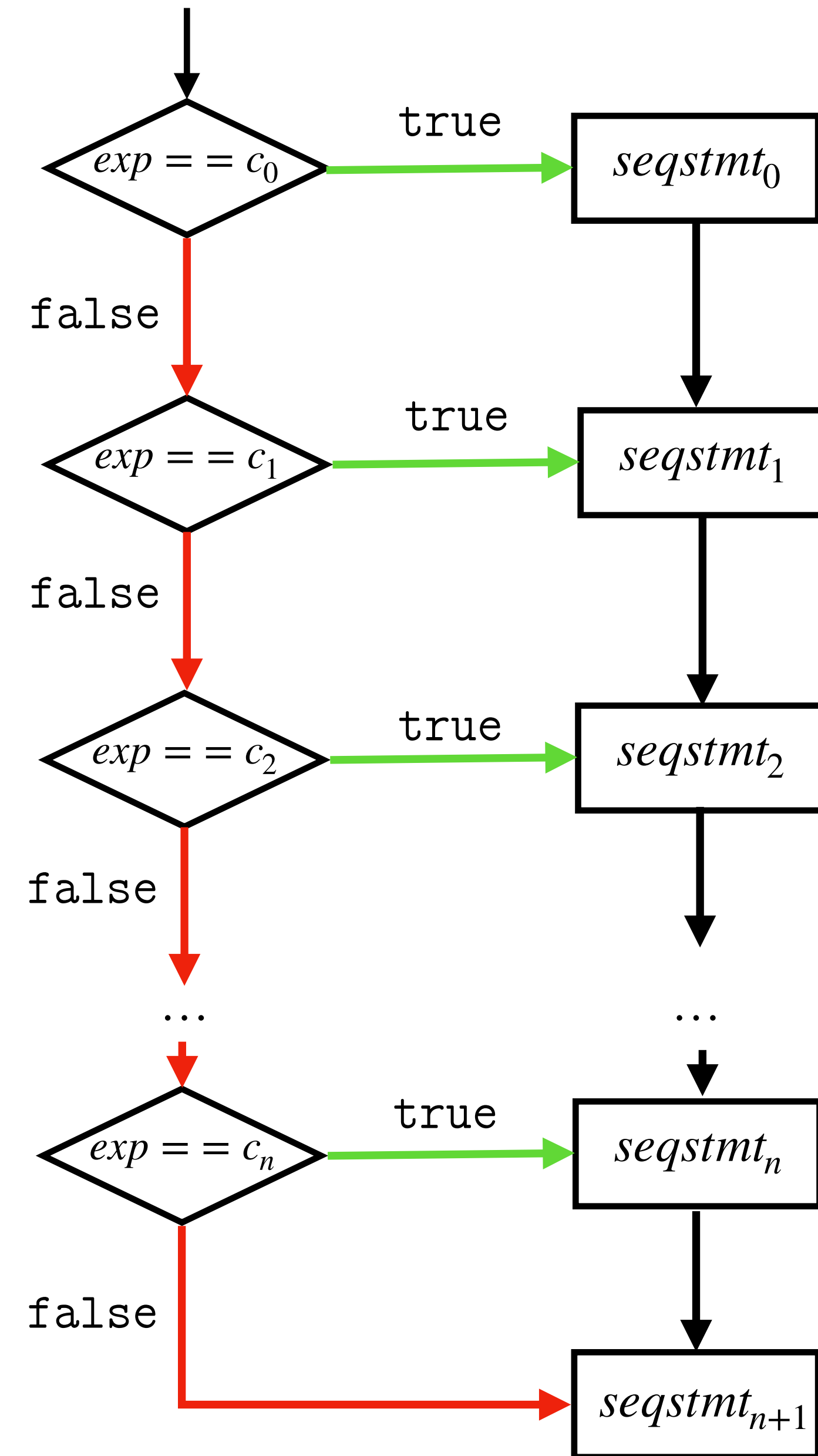
```
switch (exp) {  
  case  $c_0$  :  
    seqstmt0  
  case  $c_1$  :  
    seqstmt1  
  case  $c_2$  :  
    seqstmt2  
  ...  
  
  case  $c_n$  :  
    seqstmtn  
  default :  
    seqstmtn+1  
}
```

- se  $exp == c_i$ , per qualche  $i \in [0, n]$  esegue  $seqstmt_i, seqstmt_{i+1}, seqstmt_{i+2} \dots$ , cioè esegue **tutte** le sequenze di statement che si trovano dei case dopo il caso matchato
- se  $exp \neq c_i$ , per tutti gli  $i \in [0, n]$  esegue  $seqstmt_{n+1}$  (sequenza di statement del caso default) e **tutte** le sequenze di statement che si trovano dei case dopo il caso default

# Statement switch

## Flow-chart

```
switch (exp) {  
  case  $c_0$ :  
    seqstmt0  
  case  $c_1$ :  
    seqstmt1  
  case  $c_2$ :  
    seqstmt2  
  ...  
  case  $c_n$ :  
    seqstmtn  
  default:  
    seqstmtn+1  
}
```



# Statement switch

Selezione "esclusiva"

```
switch (exp) {  
  case c0:  
    seqstmt0 break;  
  case c1:  
    seqstmt1 break;  
  case c2:  
    seqstmt2 break;  
  ...  
  case cn:  
    seqstmtn break;  
  default:  
    seqstmtn+1  
}
```

