



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Espressioni e operatori

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE



Agenda

- Espressioni
- Operatori con side effect
- Ordine di valutazione di un'espressione



Espressioni

- Il più piccolo elemento usato per esprimere la computazione è l'**espressione**
- Un'espressione calcola un risultato a partire da uno o più operandi
- Espressioni semplici:
 - Literal: 10, 'a', 3.14, "Norah"
 - Nomi di variabili
- Espressioni semplici possono essere combinate con operatori per formare espressioni più complesse

```
int perimeter = (length + width) * 2;
```



- Un elemento molto importante nelle espressioni è l'lvalue (left value)
- Un lvalue è ciò che sta alla sinistra di un operatore di assegnamento
 - Variabili
- è ciò su cui si scrive
- Un nome di variabile è **sempre** un lvalue?

```
int length;  
length = 99;
```



Lvalue vs rvalue

- Length si riferisce:
 - Come lvalue: a un oggetto di tipo int che contiene il valore 99 – **length è il box (l'oggetto)**
 - Come rvalue: il riferimento al **valore contenuto** nell'oggetto
- In un'espressione, length può essere usato sia come lvalue che come rvalue

```
int length;
length = 99;
length /= 2;
int width;
width = length * 2;
```

int:
length: 99



Espressioni costanti

- Il software ha bisogno di molte espressioni costanti
- Il C++ permette di esprimere una costante simbolica

```
constexpr double pi = 3.14159;  
pi = 7;                      // errore!  
double c = 2 * pi * r;        // OK: sola lettura
```

- Meglio dei magic numbers / magic constants
- Meglio dei #define (macro stile C) perché le variabili costanti hanno un tipo

non ne definisce il tipo



Constexpr vs const

- Due modi per esprimere una costante
 - Constexpr: il valore è noto a tempo di compilazione
 - Const: il valore può essere noto solo a tempo di esecuzione ed è assegnato in inizializzazione

```
constexpr int max = 100;

void use (int n)
{
    constexpr int c1 = max + 7;          // OK: da costanti
    const int c2 = n + 7;                // OK

    c2 = 7;                            // errore
}
```

Side effect e valutazione delle espressioni



Operatori e side effect

- Gli operatori sono caratterizzati da:
 - 1, 2, 3 operandi
 - **Un risultato** → è sempre presente, anche se per molti operatori viene ignorato
- Alcuni operatori hanno un *side effect*
 - Modificano gli operandi su cui operano
 - Forniscono un risultato
- Es: hanno side effect `++`, `--`, `+=`, `-=`, ...



Valutazione delle espressioni

- In che ordine sono valutate le espressioni?

```
int perimeter = (length + width) * 2;
```

- Operazioni:

- Lettura di length
- Lettura di width
- Somma
- Prodotto
- Assegnamento

Quale delle due
è svolta per prima?

Hanno un ordine
obbligatorio

lo standard non lo decide,
se ne occupa il compilatore



Valutazione delle variabili

- Maggiori possibilità di **ottimizzazione** se il compilatore ha qualche grado di libertà
 - Libertà nell'ordine di valutazione delle espressioni
- Libertà del compilatore: comportamento indefinito quando non sono fissate le regole



Valutazione delle espressioni

- L'ordine di valutazione delle espressioni non è definito
 - È sbagliato usare **la stessa variabile più di una volta** se su essa sono applicati operatori con **side effect**



Valutazione delle espressioni

```
v[i] = ++i;      // Ordine di valutazione non definito:  
                 // i a sinistra è letto prima o dopo la  
                 // valutazione di ++i?  
  
v[++i] = i;      // Ordine di valutazione non definito:  
                 // Come sopra  
  
int x = ++i + ++i;    // Ordine di valutazione non definito:  
                      // lettura e incremento potrebbero  
                      // non essere consecutivi  
  
cout << ++i << ' ' << i << '\n';    // Ordine di valutazione  
                                         // non definito
```

- Nota: = è un operatore
- Anche per i suoi operandi non è definito quale dei due sia valutato per primo

Tipi che influenzano
gli operatori



Operatori << e >>

- Alcuni operatori hanno un funzionamento che dipende dai tipi degli operandi
- Abbiamo visto l'operatore << (inserimento in uno stream) **(il risultato è cout)**
- Esiste l'operatore complementare: >> (estrazione da uno stream)
- Effettuano una trasformazione dalla forma *testuale* alla rappresentazione interna delle variabili
 - Sono sensibili al tipo



Operatore di input

- Esempio: operatore >> sensibile al tipo

```
// ...

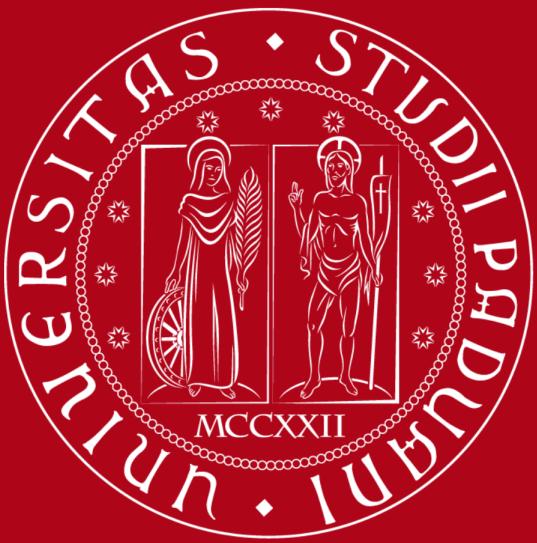
int main()
{
    std::cout << "Please enter your first name and age\n";
    string first_name;
    int age;
    std::cin >> first_name;
    std::cin >> age;
    std::cout << "Hello, " << first_name << " (age " << age <<
")\n";

    return 0;
}
```



Altri operatori sensibili al tipo

```
int count;  
std::cin >> count;  
std::string name;  
  
int c2 = count + 2;           // add integer  
std::string s2 = name + " Jr."; // append string  
  
int c3 = count - 2;           // subtracts integer  
std::string c3 = name - " Jr."; // error: not defined for  
                                // strings
```



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Espressioni e operatori

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE