



UNIVERSITÀ  
degli STUDI  
di CATANIA

# Namespace e overloading dei metodi

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

---

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: [gfarinella@dmf.unict.it](mailto:gfarinella@dmf.unict.it)

Dipartimento di Matematica e Informatica

1. I Namespace in C++
2. Overloading dei metodi in C++

# I Namespace in C++

---

In generale, un programma **C++** si compone di varie parti (modularità). In particolare:

1. **Librerie, file sorgenti e file header** sono i “primi” strumenti per implementazione della **modularità**;
2. Classi, metodi e **namespace** rappresentano strumenti più “avanzati” per la modularità;

# Namespace

**Esempio:** Gaetano e Claudio hanno implementato una funzione denominata `estrai()` che estrae un elemento da un dato vettore.

```
1 #include "gaetano_extract.h"
2 #include "claudio_extract.h"
3 //...
4 int v[] = {...};
5 int numero = estrai(v, l); //quale funzione estrai() ??
```


Includere i due file header (due prototipi!!) “confonderà” il compilatore **che informerà il programmatore che la funzione `estrai()` è stata definita due volte.**

⇒ **Name clash** o conflitto tra i nomi.

# Namespace

I namespace (C++):

- introducono un ulteriore **livello di scope**, il **più alto**.
- sono “**contenitori**” di nomi.

```
1 namespace gaetano{   
2     int *estrai(int *v, int l);  
3 }
```

Dato che lo scope della funzione `estrai()` non è più globale, **si dovrà specificare** (vedi slide successiva) il **namespace**.

In tal modo il conflitto sarà risolto, in altre parole esso **si sposta ad un livello più alto** (quello del nome del namespace!).

# Namespace

## 1-Usare **operatore risoluzione di scope**

```
int *estratto = gaetano::estrai(my_vector, l)
```

## 2-Dichiarare la singola funzione

```
using gaetano::estrai;  
//...  
int *estratto = estrai(my_vector, l);
```

## 3-Accesso all'intero namespace

```
using namespace gaetano;  
//...  
int *estratto = estrai(my_vector, l);
```

# Namespace

Accesso all'intero namespace conveniente se si usano frequentemente i nomi definiti all'interno di esso.

```
1 using namespace std;  
2 //...  
3 cout << "Hello World!" << endl;
```

VS

```
1 std::cout << "Hello World!" << endl;
```



## Esempi svolti

26\_01.cpp – namespace e metodi

26\_02.cpp – namespace e classi

## Overloading dei metodi in C++

---

# Overloading di metodi

Una famiglia di funzioni overloaded è un insieme di funzioni distinte che hanno lo stesso nome.

Esigenza di eseguire la **stessa operazione su dati di tipo differenti e/o un numero di argomenti differenti**.

Esempio (banale): operatore binario  $+$  applicato ai tipi primitivi.

$$11.7f + 3.2$$
$$24 + 7$$

La **semantica dell'operazione somma non cambia**, neppure l'operatore, tuttavia cambia il **tipo di argomenti** (float e double, int e int)

# Overloading di metodi

In cosa differiscono i seguenti prototipi?

```
void print(int x);  
void print(char c);  
void print(const char *s);  
void print(const char *s, int n);
```

...numero e/o tipo dei parametri formali.

Il seguente è overloading?

```
void print(const char *s);  
int print(const char *s);
```

NO! Per ottenere famiglia di funzioni overloaded deve cambiare lista di parametri (tipo e/o numero di essi).

# Overloading di metodi

**Domanda:** a cosa serve overloading di metodi?

Overloading è una forma di programmazione “generica” in cui una funzione con lo stesso nome viene usata con input di tipo differente.

Si consideri la seg. famiglia di funzioni overloaded

```
void print(int x);  
void print(char c);  
void print(const char *s);  
void print(double d);  
void print(long l);
```

# Overloading di metodi

Soluzione equivalente senza overloading:

```
→ void print_int(int x);  
void print_char(char c);  
void print_string(const char *s);  
void print_double(double d);  
void print_long(long l);
```

- (-) Bisogna ricordare tutti i nomi di funzione (print\_char, print\_string, etc).
- (-) Forza il programmatore a ragionare continuamente sui tipi dei parametri attuali.

# Overloading di metodi

NB: Il programmatore dovrebbe invece concentrarsi sulle **operazioni, non sul tipo dei dati** (programmazione generica).

```
print_string(" Hello World! ");  
print_double(1.23456789);  
print_long(123456789123);
```

VS

```
print(" Hello World! ");  
print(1.23456789);  
print(123456789123);
```

# Overloading di metodi

Overloading di metodi vs argomenti standard.

Alcuni casi **non possono essere gestiti mediante gli argomenti standard.**

```
double combine(int x, int y, int z){  
    if(z==0) return (x+y)/2.0;  
    else return z * (x+y)/2.0;  
}
```

```
double combine(int x, int y){  
    return (x - y) / ((double) (x+y));  
}
```

Cambia (parzialmente) anche semantica funzione.



# Overloading di metodi: Risoluzione

**Risoluzione:** il compilatore associa l'invocazione della funzione ad una delle funzioni overloaded.

```
void f(int a, double b, char c);  
void f(int a, double b);
```

```
f(5, 5.4, 'Z'); // call to f(int a, double b, char c);  
f(5, 7.8); // call to f(int a, double b);
```

Caso “semplice”, la discriminante è il numero degli argomenti.

# Overloading di metodi: Risoluzione

Cosa fa il compilatore se non esiste corrispondenza esatta tra tipi dei parametri attuali e quelli dei parametri formali?

```
void print(int);  
void print(char);  
void print(const char *);  
void print(double);  
void print(long);
```

```
→ print(4); // OK, invoca print(int)  
✓ print("pippo"); //OK, invoca print(const char *s)  
→ print(1.4f); // conversione: float → double  
short k=2; print(k); // promozione: short → int
```

# Overloading di metodi: Risoluzione

Casi di ambiguità.

Prototipi

```
void setData(int); // 1  
✓ void setData(int, int); // 2 ←  
✓ void setData(double, double); // 3
```

Associazioni del compilatore:

```
→ setData(10.2); // 1 (conversione double → int)  
setData(10, 8); // 2 ←  
→ setData(3.4, 9); // ambigua: 2 o 3?  
→ setData(9, 3.4); // ambigua: 2 o 3?
```

## Algoritmo di risoluzione.

1. **Corrispondenza esatta** degli argomenti oppure conversioni “banali” (nome array  $\rightarrow$  puntatore,  $\langle \text{type} \rangle \rightarrow \text{const } \langle \text{type} \rangle$ )
2. **promozioni** tra interi (e.g. short  $\rightarrow$  int, bool  $\rightarrow$  int) e tra numeri in virgola mobile (float  $\rightarrow$  double)
3. **conversioni standard** (e.g. double  $\rightarrow$  int, etc.)
4. **conversioni definite dall'utente** (int ClasseX quando esiste un costruttore ClasseX(int))
5. corrispondenza con **ellipsis**..

Esempi svolti

26\_03.cpp

## Ellipsis

C/C++ offrono la possibilità di dichiarare funzioni che prendono un numero variabile di argomenti.

```
int sum(int NCount, ...); //ellipsis
//...
int main(){
    int mysum = sum(2,4,5,1,2,3,9); //OK
}
```

# Uso di Ellipsis

Definire il corpo di un metodo che fa uso di ellipsis.

```
1  #include <cstdarg>
2
3  int sum(int nCount, ...){
4      va_list list;
5
6      va_start(nCount, list);
7
8      for(int i=0; i<nCount; i++)
9          sum += va_arg(list, int); // “consuma” gli argomenti
10 }
```

va\_list, va\_arg, va\_start sono nomi (strutture e macro)  
definite nello header cstdarg

Esempi svolti

26\_04.cpp