Le funzioni in C++

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Definizioni di Funzioni (Quasi come il C)

Le funzioni si definiscono e si invocano come in C

```
unsigned long int fattoriale(unsigned int n) {
  if (n<2) return 1;

  return n*fattoriale(n-1);
}
```

[Vedi Cap 4.5 di PPP]

Funzioni e il tipo di output `auto` (C++14)

Le funzioni possono avere `auto` come tipo di uscita

```
auto fattoriale(unsigned int n) {
  if (n<2) return 1;

  return n*fattoriale(n-1);
}
```

Il compilatore deduce il tipo restituito

Dichiarazioni vs Definizioni (Come in C)

Prima di invocare una funzione, ne va *dichiarata la firma*

```
void prova(int a);              // questa è la dichiarazione

int main() {
    ...
    prova(5);                   // questa è l'invocazione
    ...
}

void prova(int a) {             // questa è la definizione
    ...
}
```

Passaggio di Parametri per Valore

In C il passaggio di parametri è **sempre** per valore.

```
void assegna_1(int* p, int v) { *p = v; }

assegna_1(&x, 1);   // La chiamata assegna a 'x' il valore 1
```

Il parametro attuale deve essere un valore del tipo corretto, e.g., `&x` è un espressione del tipo `int*`

Il C++ ammette il passaggio per valore e ne introduce un altro tipo

Passaggio di Parametri per Riferimento

Il C++ ne ammette 2 tipi

- *Per riferimento "semplice"*

```
void assegna_2(int &r, int v) { r = v; }

assegna_2(x, 5);   // La chiamata assegna a 'x' il valore 1
```

- *Per riferimento di un rvalue* o *valore a destra*

```
void assegna_3(int &r, int &&v) { r = v; }

assegna_3(x, 5+7);   // il parametro attuale è la valutazione di un'espressione
```

I parametri `const`

Quando non sono modificati, è meglio dichiararli `const`

```
void assegna_4(int &p, const int v) {
    *p = v;

    v++;   // questa linea di codice genera un errore
}
```

Questo consente al compilatore di ottimizzare il codice e usare un alias al posto del parametro formale

Overloading (C++)

Funzioni con parametri diversi possono avere lo stesso nome

```
int media(const int &a, const int &b) { return (a+b)/2; }

double media(const double &a, const double &b) { return (a+b)/2; }

double media(const float &a, const char b)  { return (a+b)/2; }
```

Il compilatore deduce quale invocare usando i tipi dei parametri

```
media(4.2, 7.2);  // i parametri sono double --> invoca la seconda `media`
```

Overloading (Con'd)

Non sono consentite funzioni con:

- lo stesso nome
- gli stessi parametri
- diverso tipo restituito

```
int media(const int &a, const int &b) { ... }     // OK

double media(const int &a, const int &b) { ... }  // errore
```

Puntatori a Funzioni

Possono essere usate per riferirsi a funzioni

```
int func1(const double a);
int func2(const int a);
...

int (*f)(const double);     // 'f' è un puntatore a funzioni

f = func1;                  // possiamo assegnalo con puntatori a funzioni
                            // con una firma consistente al tipo di 'f'

f(9.2);                     // possiamo invocale la funzione puntata da 'f'

// f = func2;               // ERRORE: 'func2' ha un tipo diverso da 'f'
```

Puntatori a Funzioni: `auto` e `decltype`

Possono essere usate anche con i puntatori a funzioni

```
int func1(const double a);
...

auto f = func1;                  // 'auto' funziona anche con
                                 // i puntatori a funzioni
f(2.1);

decltype(&func1) f2 = func1;     // decltype può essere usato
                                 // anche per determinare il
                                 // tipo delle funzioni
f2(3.2)
```

Funzioni Template (C++)

Possiamo parametrizzare i tipi nelle funzioni

```
template<typename T, typename T2>
T media(const T &a, const T2 &b) {
  T tmp = a+b;

  return tmp/2;
}

media(2,3);      // i parametri del template vengono dedotti
media(0, 2.3);   // dai parametri della funzione
```

[Vedi Cap 19.3 di PPP]

Funzioni Template (Cont'd)

I tipo possono essere specificati in fase di invocazione

```
template<typename T, typename T2>
T media(const int &a, const T2 &b) {
  T tmp = a+b;

  return tmp/2;
}

media<unsigned int, double>(2,3.2);
```

Specializzazione dei Template

È possibile specializzare le funzioni in base al parametro

```
template<typename T>          // questa è la definizione del template
T sorpresa(const T& a) {
  return a/2;
}

template<>                    // questa è la specializzazione per T=int
int sorpresa<int>(const int& a) {
  return 2*a;
}
```

Template e valutazioni al *compilation time*

I parametri del template possono essere usati come i parametri

```
template<unsigned i>                          // questa è una funzione template il
constexpr unsigned long int fib() {           // cui parametro è un `unsigned int`
  return fib<i-1>() + fib<i-2>();
}

template<>
constexpr unsigned long int fib<0>() {        // questa è una specializzazione
  return 0;                                   // di `fib`
}
```

Template e valutazioni al *compilation time* (Cont'd)

```
template<>
constexpr unsigned long int fib<0>() {    // questa è un'altra
  return 1;                               // specializzazione di `fib`
}

std::cout << fib<5>() << std::endl;
```

In questo caso, la valutazione del parametro viene fatta al tempo di compilazione, ma la somma durante l'esecuzione