Università della Calabria

Corso di Laurea in Ingegneria Elettronica

Programmazione Orientata agli Oggetti

A.A. 2006/2007

Funzioni

Paolo Trunfio *

* DEIS, Università della Calabria – http://si.deis.unical.it/~trunfio

Funzioni

La definizione di una funzione segue il seguente schema generale:

```
tipo_rit nome_funz (tipo_par1 nome_par1, … , tipo_parN nome_parN) {
  istruzioni
}
```

Esempi:

```
int somma (int a, int b) {
  int x = a + b;
  return x;
}

void stampa (double x, int n) {
  for (int i = 0; i < n; i++)
    cout << x << '\t';
}
```

Funzioni

Le funzioni devono essere *definite* prima di essere utilizzate:

```
#include <iostream>
using namespace std;

int sommaNumeri(int k) {
  int somma = 0;
  for (int i = 1; i <= k; i++)
    somma += leggiNumero();  // Errore
  return somma;
}

int leggiNumero() {
  cout << "Inserisci numero: ";
  int a;
  cin >> a;
  return a;
}

int main() {
  const int n = 5;
  int s = sommaNumeri(n);
  cout << "Somma = " << s << '\n';
  return 0;
}
```

```
#include <iostream>
using namespace std;

int leggiNumero() {
  cout << "Inserisci numero: ";
  int a;
  cin >> a;
  return a;
}

int sommaNumeri(int k) {
  int somma = 0;
  for (int i = 1; i <= k; i++)
    somma += leggiNumero();  // OK
  return somma;
}

int main() {
  const int n = 5;
  int s = sommaNumeri(n);
  cout << "Somma = " << s << '\n';
  return 0;
}
```


Dichiarazione di funzioni

E' possibile invocare una funzione *definita* successivamente, purchè sia *dichiarata* prima del punto in cui viene invocata.
Nell'esempio precedente si potrebbe scrivere:

```
int leggiNumero();  // dichiarazione della funzione leggiNumero

int sommaNumeri(int k) {
  int somma = 0;
  for (int i = 1; i <= k; i++)
    somma += leggiNumero();  // OK, la funzione è stata dichiarata
  return somma;
}

int leggiNumero() { // definizione della funzione leggiNumero
  cout << "Inserisci numero: ";
  int a;
  cin >> a;
  return a;
}
```

Prototipi di funzioni

La dichiarazione di una funzione si specifica mediante un cosiddetto "prototipo di funzione".
Il prototipo di una funzione è uguale alla definizione di una funzione, ma è privo del corpo (ovvero non specifica le istruzioni della funzione).

Esempio:

```
int prodotto (int a, int b);  // prototipo (dichiarazione) della funzione

int prodotto (int a, int b)  // definizione della funzione
{
  return a * b;
}
```

E' possibile omettere i nomi dei parametri nel prototipo di una funzione. Il prototipo precedente può essere scritto in modo equivalente come segue:

```
int prodotto (int, int);
```


Risoluzione di visibilità

Variabili con lo stesso nome possono essere dichiarate con visibilità (*scope*) diversa:

```
int x = 0;  // global scope

void f() {
 int x = 9;  // function scope: nasconde la x con global scope
 for (int x = 1; x <= 5; x++) { // block scope: nasconde la x con function scope
  cout << "x = " << x << endl;  // x = 1...5
 }
 cout << "x = " << x << endl;  // x = 9
 cout << "x = " << ::x << endl;  // x = 0
}
```

La coppia di simboli **::** costituisce l'operatore di risoluzione di visibilità.

Passaggio di parametri per copia

```
#include <iostream>
using namespace std;

void f1 (int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int x = 5, y = 9;
  cout << x << " " << y << endl;  // 5 9
  f1 (x, y);
  cout << x << " " << y << endl;  // 5 9
}
```


Passaggio di parametri per riferimento

```
#include <iostream>
using namespace std;

void f2 (int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int x = 5, y = 9;
  cout << x << " " << y << endl;  // 5 9
  f2 (x, y);
  cout << x << " " << y << endl;  // 9 5
}
```

Passaggio di parametri puntatori

```
#include <iostream>
using namespace std;

void f3 (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  int x = 5, y = 9;
  cout << x << " " << y << endl;  // 5 9
  f3 (&x, &y);
  cout << x << " " << y << endl;  // 9 5
}
```


Passaggio di parametri per riferimento a costante

```
#include <iostream>
#include <string>
using namespace std;

int contaSpazi (const string &s) {  // non è possibile modificare s
  int spazi = 0;
  for (int i=0; i<s.size(); i++)
    if (s[i] == ' ')
      spazi++;
  return spazi;
}

int main() {
  string cdl = "Ingegneria Elettronica";
  int spazi = contaSpazi (cdl);
  cout << spazi << endl;
}
```

Passaggio di parametri puntatori a costante

```
#include <iostream>
#include <string>
using namespace std;

int contaSpazi (const string *s) {  // non è possibile modificare s
  int spazi = 0;
  for (int i=0; i<(*s).size(); i++)
    if ((*s)[i] == ' ')
      spazi++;
  return spazi;
}

int main() {
  string cdl = "Ingegneria Elettronica";
  int spazi = contaSpazi (&cdl);
  cout << spazi << endl;
}
```


Passaggio di parametri array

Il passaggio di un array ad una funzione equivale a passare un puntatore al primo elemento dell'array.

Il passaggio per copia degli array non è supportato per motivi di efficienza.

```
void azzera (int v[], int dim) {
  for (int i=0; i<dim; i++)
    v[i] = 0;
}
```

La funzione precedente può essere scritta in modo equivalente come segue:

```
void azzera (int *v, int dim) {
  for (int i=0; i<dim; i++)
    v[i] = 0;
}
```

Si noti che la funzione *azzera* modifica l'array originale.

Passaggio di parametri array

Per garantire l'assenza di modifiche ad un parametro array all'interno di una funzione è possibile utilizzare la parola *const* come di consueto.

```
int somma (const int v[], int dim) {  // int somma (const int *v, int dim)
  int s = 0;
  for (int i=0; i<dim; i++)
    s += v[i];
  return s;
}

int main() {
  const int dim = 5;
  int *a = new int[dim];
  azzera(a, dim);
  int b[dim] = {1,2,4,8,16};
  cout << somma (a, dim) << " " << somma (b, dim) << endl;  // 0  31
  delete[] a;
}
```


Passaggio di array bidimensionali dinamici

Il metodo seguente è valido per array bidimensionali *dinamici*:

```
int somma (int **m, int nr, int nc) {
  int s = 0;
  for (int i=0; i<nr; i++)
    for (int j=0; j<nc; j++)
      s += m[i][j];
  return s;
}
```

A differenza del caso monodimensionale, non è possibile utilizzare le parentesi quadre al posto degli asterischi nella dichiarazione dei parametri:

```
int somma (int [][]m, int nr, int nc)  // Errore
```

Passaggio di array bidimensionali non dinamici

Per passare un array bidimensionale *non dinamico* ad una funzione è necessario specificare il numero di colonne:

```
int somma (int m[][3], int nr, int nc) {
  int s = 0;
  for (int i=0; i<nr; i++)
    for (int j=0; j<nc; j++)
      s += m[i][j];
  return s;
}

int main() {
 int m[2][3] = { {1,2,3}, {4,5,6} };
 cout << somma(m, 2, 3) << endl;  // 21
}
```

In generale, quando si passa un array multidimensionale ad una funzione è necessario specificare il valore per tutte le dimensioni successive alla prima.

Ritorno di una copia

```
#include <iostream>
#include <string>
using namespace std;

int quadrato (int n) {
  int q = n*n;
  return q;
}

int main() {
  int a = 4;
  int b = quadrato (a);
  cout << b << endl;
}
```

Ritorno di un riferimento

Il ritorno di un riferimento deve essere usato con cautela. In particolare, restituire il riferimento ad una variabile locale è scorretto in quanto la memoria associata alla funzione (e quindi alla variabile locale) viene rilasciata automaticamente al termine della funzione stessa.

```
int& quadrato (int n) {
  int q = n*n;
  return q;  // scorretto: restituisce il riferimento ad una variabile locale
}

int main() {
  int a = 4;
  int &b = quadrato(a);  // il valore di b può cambiare imprevedibilmente
  cout << b << endl;
}
```


Ritorno di un riferimento

Il ritorno per riferimento può essere utilizzato correttamente se l'oggetto di cui si restituisce l'indirizzo non è memorizzato nell'area dati della funzione:

```
string& concatena (const string &a, const string &b) {
  string *c = new string(a+b);
  return *c;  // ok: la stringa è memorizzata nell'heap
}

int main() {
  string a = "UNI";
  string b = "CAL";
  string &c = concatena (a, b);
  cout << c << endl;  // UNICAL
  string *d = &c;
  delete d;
}
```

Nota: in questo esempio il ritorno per copia sarebbe stato più opportuno.

Ritorno di un riferimento

Se una funzione restituisce il riferimento ad una variabile, è possibile modificare il valore della variabile originale:

```
int a = 5;  // variabile globale

int& f() {
  return a;
}

int main() {
  int &b = f();
  b = 10;  // modifica la variabile a
  int &c = f();
  cout << c << endl;  // 10
  cout << a << endl;  // 10
}
```

In generale si suggerisce di evitare il ritorno di riferimenti o puntatori a variabili globali o statiche.

Ritorno di un riferimento a costante

Per impedire di modificare la variabile originale, pur mantenendo il ritorno per riferimento, è possibile restituire un riferimento a costante:

```
int a = 5;  // variabile globale

const int& f() {
  return a;
}

int main() {
  // int &b = f();  // errore
  const int &b = f();  // ok
  // b = 10;  // errore
  cout << b << endl;  // 5
}
```

Il ritorno di un riferimento a costante è generalmente utilizzato nelle funzioni delle classi per accedere in modo efficiente ed in sola lettura alle variabili d'istanza.

Ritorno di un puntatore

Il ritorno di un puntatore richiede la stessa cautela del ritorno per riferimento. In particolare, è scorretto restituire il puntatore ad una variabile locale:

```
int* quadrato (int n) {
  int q = n*n;
  return &q;  // scorretto: restituisce l'indirizzo di una variabile locale
}

int main() {
  int a = 4;
  int *b = quadrato(a);  // il valore puntato da b può cambiare imprevedibilmente
  cout << *b << endl;
}
```


Ritorno di un puntatore

Come per il ritorno per riferimento, è corretto restituire un puntatore se la locazione di memoria puntata non è memorizzata nell'area dati della funzione:

```
string* concatena (const string &a, const string &b) {
  string *c = new string(a+b);
  return c;  // ok: la stringa è memorizzata nell'heap
}

int main() {
  string a = "UNI";
  string b = "CAL";
  string *c = concatena (a, b);
  cout << *c << endl;  // UNICAL
  delete c;
}
```

Ritorno di un puntatore a costante

Il ritorno di un puntatore a costante è simile al ritorno di un riferimento a costante:

```
string a = "Ingegneria";

const string* f() {
  return &a;
}

int main() {
  // string *b = f();  // non ammesso
  const string *b = f();  // ok
  // (*b)[0] = 'i';  // non ammesso
  cout << *b << endl;  // Ingegneria
}
```


Ritorno di un array

```
#include <iostream>
using namespace std;

int* leggiArray (int dim) {
  int *v = new int[dim];
  for (int i=0; i<dim; i++) {
    cout << "Elemento " << i << " = ";
    cin >> v[i];
  }
  return v;
}

int main() {
  const int dim = 5;
  int *a = leggiArray (dim);
  for (int i=0; i<dim; i++)
    cout << a[i] << " ";
  delete[] a;
}
```

Overload di funzioni

Più funzioni possono condividere lo stesso nome purchè differiscano nel tipo e/o nel numero dei parametri.

```
void f(int a, int b) {
  cout << a << " + " << b << " = " << a + b << endl;
}
void f(int a) {
  cout << a << " + 1 = " << a + 1 << endl;
}
void f(double a, double b) {
  cout << a << " + " << b << " = " << a + b << endl;
}
int main() {
  int a = 6, b = 4;
  double x = 3.6, y = 2.1;
  f(a,b);  // 6 + 4 = 10
  f(a);    // 6 + 1 = 7
  f(x,y);  // 3.6 + 2.1 = 5.7
  f(x);       // Warning: 3 + 1 = 4
  // f(a,x);  // Errore: chiamata ambigua
}
```


Parametri di default

E' possibile specificare il valore di default che un parametro deve assumere nel caso in cui non sia passato all'atto dell'invocazione della funzione:

```
void f(int a, int b = 1) {
  cout << a << " + " << b << " = " << a + b << endl;
}

int main() {
  int a = 6, b = 4;
  f(a,b);  // 6 + 4 = 10
  f(a);    // 6 + 1 = 7
}
```

I parametri di default devono apparire alla destra dei parametri che non specificano un valore di default:

```
int f1(double x, int y = 5, double z = 3.14);  // OK

int f2(double x = 3.5, int y, double z = 7.5);  // Errore
```

Funzioni inline

Il modificatore *inline* chiede al compilatore di espandere in linea il codice della funzione in modo da renderne più efficiente l'esecuzione.

```
inline int leggiIntero (const string &messaggio) {
  cout << messaggio;
  int x;
  cin >> x;
  return x;
}

int main() {
  int s = 0;
  for (int i=1; i<=10; i++)
    s += leggiIntero("Inserisci un intero: ");
  cout << "Somma = " << s << endl;
}
```

Non è possibile definire funzioni inline "complesse" (il limite di complessità dipende dal compilatore).