INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento
A.A. 2003/2004
II Bimestre

Marco Roveri
roveri@irst.itc.it

Funzioni e Procedure
(Materiale preso ed adattato da materiale di Marco Benedetti)

Funzioni e Procedure

- Motivazioni:
  - Quando si scrivono programmi di una certa complessità, per rendere il programma "leggibile" occorre strutturarlo in parti modulari e riutilizzabili. Ad esempio, se si deve eseguire ripetutamente una serie di istruzioni complesse, risulta insoddisfacente ripetere più volte la stessa parte di codice.
- Per sopperire a questi inconvenienti, risulta utile strutturare il codice in "*procedure*" e "*funzioni*".

Funzioni

- Da un punto di vista del programmatore una funzione è un *segmento di codice* adibito ad un compito *specifico*.
  - Calcolo del fattoriale di un numero.
- Il *compito* di una funzione è quello di calcolare un risultato (detto *valore di ritorno*), partendo da specifici *valori di ingresso (parametri)*.
  - x = fattoriale(10)
- Per ogni funzione *deve* essere specificato a priori:
  - il *tipo* del valore di ritorno (ovvero il codominio);
  - il *tipo* e la *quantità* dei parametri (ovvero il dominio).

Procedure

- Molto spesso ci troviamo a definire delle "funzioni" che non restituiscono nulla, però effettuano delle operazioni che si ripetono (e.g. stampa degli elementi di un array, ed in generale routines che stampano o forniscono viste su strutture).
- In questo caso non possiamo parlare di funzioni nel senso stretto del termine, in quanto non viene ritornato nessun valore, ovvero il codominio è vuoto.
- In questo caso parliamo di *procedure.*
- Il C++ vede le procedure come particolari tipi di funzione, dove il valore di ritorno non esiste (si usa il tipo **void**).

Le funzioni C++

- *Dichiarazione*: dove viene dato il *prototipo*:
  - *Nome della funzione;*
  - *Numero e tipo dei parametri;*
  - *Tipo del valore di ritorno;*
- *Definizione*: si specifica come una funzione precedentemente dichiarata calcola il valore di ritorno in base ai *parametri formali*. Rappresenta il frammento di codice che rappresenta il *corpo (body)* della funzione stessa.
- *Chiamata (o invocazione)*: si utilizza una funzione già dichiarata, passando i *parametri attuali*, coerenti in tipo e numero con i parametri formali; si utilizza poi il valore di ritorno opportunamente.

Funzioni: Dichiarazione e Definizione

```
// Dichiarazione: una funzione dal nome  isprime che prende
// un intero e ritorna un booleano che è true se l'intero è un
// numero primo, false altrimenti.
bool isprime(int);

// Definizione:
bool isprime(int n) {
  int j = 2;

  while( ((n % j) != 0) && (j < n)) j++;
  return (j == n);
}
```

Funzioni: invocazione

```
int main() {
  int n;
  for(int i = 1; i < 10; i++) {
    cin >> n;
    if (isprime(n) == true) {
      cout << "Il numero " << n << " e' un numero primo" << endl;
    }
    else {
      cout << "Il numero " << n << " non e' un numero primo" << endl;
    }
}
```


Funzioni: considerazioni

- Le funzioni possono essere *definite una sola volta*.
- Le funzioni possono essere *dichiarate più di una volta*.
- Le funzioni possono essere *invocate un numero qualunque di volte*, da un qualunque punto di un programma sia visibile la definizione della funzione (quindi può anche invocare se stessa, funzioni *ricorsive*).
- Nel corpo di funzioni deve esserci almeno una occorrenza dell'istruzione **return <expr>**, dove <expr> deve essere una espressione dello stesso tipo del valore di ritorno della funzione.
  - Nel caso di procedure <expr> deve essere omesso.
  - Inoltre, il vincolo di avere almeno una occorrenza dell'istruzione return può essere rilassato.
- Le funzioni possono apparire in espressioni, a patto che il tipo della funzione sia compatibile con le regole di ben fondatezza dell'espressione in cui occorre.
- Le procedure, invece, non possono apparire in espressioni, ma al contrario occorrono nel codice come un'*istruzione.*

Funzioni: variabili locali

- Per ogni invocazione di una funzione viene creato uno spazio di memoria, detto *record di attivazione*, dove la funzione memorizza le sue variabili locali.
- Questo spazio vive per il tempo di vita della funzione, quindi tutte le variabili dichiarate all'interno della funzione non sono visibili all'esterno della funzione stessa.
- Le informazioni con l'ambiente esterno si scambiano solo attraverso i parametri e il valore di ritorno.
- I parametri formali sono assimilabili a delle variabili locali. I parametri formali sono inizializzati con i valori usati per i parametri attuali durante l'invocazione.

Funzioni: passaggio per valore

- I parametri formali sono assimilabili a variabili locali all'interno del corpo di una funzione.
- In questo modo, una eventuale modifica al valore di una variabile parametro non ha nessun effetto sul valore del corrispondente parametro attuale.
- Questa modalità di passaggio dei parametri è detta *passaggio per valore*, per evidenziare il fatto che è il *valore* ad essere passato e *non la variabile che lo contiene*.
- Il record di attivazione, *contiene una copia del valore* della variabile corrispondente al parametro attuale.

Funzioni: passaggio per valore

```
void fakeswap(int x, int y) {
  int z = x;
  x = y;
  y = z;
  cout << "x = " << x << endl << "y = " << y << endl;
}

int main() {
 int x = 10, y = 0;
 cout << "x = " << x << endl << "y = " << y << endl;
 fakeswap(x,y);
 cout << "x = " << x << endl << "y = " << y << endl;
 }
```


Funzioni: passaggio per riferimento

- Il C++, oltre al passaggio per valore, supporta il cosiddetto *passaggio per riferimento*.
- In questo caso il record di attivazione della funzione *non* contiene una copia del valore, ma si riferisce direttamente alla locazione di memoria corrispondente alla variabile considerata (nella lezione su puntatori e memoria questo sarà più chiaro).
- Una qualsiasi variazione al valore della variabile locale si ripercuoterà quindi in una "equivalente" (si agisce sulla stessa locazione di memoria) variazione della variabile corrispondente al parametro attuale.

Funzioni: passaggio per riferimento

- Quando non specificato, il compilatore assume che la variabile venga passata per valore.
- Se si vuole passare un parametro per riferimento, occorre informare il compilatore.
- Questo si effettua interponendo tra il tipo del paramero formale e il suo nome la direttiva "*&*".
- Esempio:

  void swap(int **&** x, int **&** y);

13

Funzioni: passaggio per riferimento

```
void swap(int & x, int & y) {
 int z = x;
 x = y;
 y = z;
 cout << "x = " << x << endl << "y = " << y << endl;
}

int main() {
 int x = 10, y = 0;
 cout << "x = " << x << endl << "y = " << y << endl;
 swap(x,y);
 cout << "x = " << x << endl << "y = " << y << endl;
}
```

14

Passaggio per riferimento

- Quando utilizzarlo:
  - Se si vuole fare *side-effect* sui parametri attuali di una funzione (e.g. la funzione swap()).
  - Quando i parametri sono utilizzati non solo come parametri di ingresso, ma anche come parametri di uscita (e.g. funzione che deve restituire due valori).
  - Quando la dimensione (occupazione di memoria) del parametro attuale è "grande" e per questioni di efficienza non si vuole "perdere" del tempo nel copiare nel record di attivazione il parametro attuale.

15

Funzioni: passaggio per riferimento

- Quando non utilizzarli:
  - Quando una caratterizzazione funzionale è possibile (migliore leggibilità del codice).
  - Quando il parametro attuale non è una variabile ma una espressione complessa (in tal caso non c'è una vera e propria variabile).
- L'utilizzo del passaggio per riferimento può essere causa di errori involontari, e.g. più identificatori distinti si riferiscono allo stesso oggetto:
  - Esempio:

```
somma(x, x, 10);
….
void somma(int & x, int &y, int z) {
 x += z;  y += z; }
```

16

Funzioni: Passaggio di array

- Gli array obbediscono a delle regole leggermente diverse.
- L'eccezione è dovuta al fatto che il C++ deriva dal C, dove la distinzione tra passaggio per valore e per riferimento non è perfettamente gestita.
- Per le funzioni e gli array le seguenti considerazioni possono essere formulate:
  - In generale una funzione *non* può restituire un array (vedremo però che in alcuni casi è possibile).
  - Quando si passa un array si sta *implicitamente* passando il contenuto dell'array per *riferimento* e dunque la funzione *può modificarlo*.

17

Funzioni: Passaggio di array

- La dichiarazione di una funzione che accetta un array (monodimensionale) di interi è ad esempio:

  type function_name(int []);

  Mentre la definizione sarà della forma:

  type function_name(int A[]) { … }
- Non è dunque necessario specificare la dimensione dell'array nel tipo del parametro.
  - Tipicamente però si usa un parametro addizionale che rappresenta la dimensione dell'array, sfruttando la cosniderazione che il chiamante ha tendenzialmente conoscenza della dimensione dell'array.

    type function_name(int A[], int N) { … }
- Quando si passano array multidimensionali, la dimensione deve invece essere data esplicitamente nel tipo del parametro:

  type function_name(int [10][10]);

  type function_name(int A[10][10]) { … }

18

Funzioni Ricorsive

- Dal punto di vista sintattico, siamo in presenza di una funzione ricorsiva quando **all'interno della definizione di una funzione compaiono una o più invocazioni alla funzione che si sta definendo**; in altre parole, la funzione "ricorre a se stessa" per svolgere il proprio compito;
- Le funzioni ricorsive risultano estremamente comode nella codifica di tutti gli algoritmi modellati su un procedimento di soluzione induttivo (nel corso vedremo diversi usi di funzioni ricorsive, negli algoritmi di ordinamento, in funzioni per attraversare od operare su alberi, grafi,…) .

Esempio: fattoriale

```
int  fattoriale(int N) {
  if (N == 0)
     return 1;
  else
     return N * fattoriale(N – 1);
}
```


Esempio: fibonacci

- Formulazione matematica:
  fibonacci(0) = 1
  fibonacci(1) = 1
  fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)
- Una possibile implementazione:

```
int fibonacci(int N) {
  if ((N == 0) || (N == 1))
     return 1;
  else
     return fibonacci(N – 1) + fibonacci(N – 2);
}
```


Funzioni ricorsive

- Vantaggio principale:
  - Intuitive per definire particolari tipi di funzioni che sono naturalmente ricorsive.
- Svantaggio principale:
  - Molto spesso le versioni ricorsive sono poco efficienti. Ogni invocazioni richiede la allocazione di un record di attivazione.
  - Siccome l'area dati dedicata allo stack di attivazione è limitato, può accadere che si presenti il cosiddetto problema dello *stack overflow* (superamento della dimensione dello stack).
  - Per sopperire a questi problemi alcuni compilatori automaticamente trasformano funzioni ricorsive in equivalenti funzioni iterative, in cui si richiede un solo record di attivazione.

Funzioni ricorsive

- Il fattoriale iterativo diventa:

```
int fattoriale(int N) {
 int r = 1;
 for( ; N > 1; N--)
   r = r * N;
 return r;
}
```


Esercizio:

- Scrivere la funzione di fibonacci iterativa.