Procedure e funzioni

In linguaggio C

Procedura

- Una procedura può essere rappresentata come una macchina in grado di eseguire un certo compito quando attivata.
- In un primo tempo la macchina deve essere costruita: la costruzione della macchina può essere paragonata con la **dichiarazione e definizione** della procedura.
- La macchina viene poi attivata per eseguire il suo compito: può essere attivata più volte e tutte le volte ritorna ad eseguire il compito per cui è stata costruita. L'avviamento della macchina può essere paragonato all'**esecuzione** della procedura.

Procedure in C

- In linguaggio C / C++ non esistono procedure ma solo funzioni. Le procedure possono essere realizzate mediante funzioni che non restituiscono valori cioè funzioni **`void`**.
- La dichiarazione di una procedura deve essere inserita prima della sua esecuzione.
- Utilizzando i prototipi è possibile definire le procedure dopo il programma principale.
- L'esecuzione di una procedura termina quando si raggiunge la sua ultima istruzione o si incontra l'istruzione **`return`**.

Esempio di procedura in C

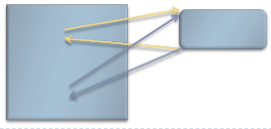
```
#include <iostream>

using namespace std;

void stampaLogo()           // dichiarazione e definizione della procedura
{
    cout<<endl;
    cout<<"*********************************"<<endl;
    cout<<"*** Classe III   Informatica ***"<<endl;
    cout<<"***  Itis Leonardo da Vinci  ***"<<endl;
    cout<<"*********************************"<<endl;
}


int main()
{
    stampaLogo();           // chiamata della procedura
    cout<<endl<<"Programma di prova"<<endl;
    stampaLogo();           // chiamata della procedura
}
```

Esempio con prototipo

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

void stampaLogo();                // dichiarazione (prototipo)

int main()
{
    stampaLogo();           // chiamata della procedura
    cout<<endl<<"Programma di prova"<<endl;
    stampaLogo();           // chiamata della procedura
}
void stampaLogo()           // definizione della procedura
{
    cout<<endl;
    cout<<"*********************************"<<endl;
    cout<<"*** Classe III   Informatica ***"<<endl;
    cout<<"***  Itis Leonardo da Vinci  ***"<<endl;
    cout<<"*********************************"<<endl;
}
```

Esecuzione del codice

- L'esecuzione di un programma C inizia dalla prima istruzione di main (anche main è una funzione).
- La chiamata di una funzione provoca l'interruzione momentanea dell'esecuzione del codice, l'esecuzione del codice della funzione e, al termina dell'esecuzione di questa, la ripresa del codice inizialmente sospeso



Scambio di informazioni fra programma e procedura

- Nell'esempio precedente il programma e la procedura non avevano la necessità di scambiarsi informazioni.
- In generale è invece necessario uno scambio di informazioni fra programma e procedura (o fra varie procedure)
- La dichiarazione di una variabile "fuori" dal programma e dalla procedura (variabile globale) permette ad entrambi di "vederla" quindi di scambiarsi, attraverso questa, informazioni.

Un esempio con variabili globali

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

int x;                // variabile globale visibile sia dalla procedura che dal main
int cub;              // variabile globale visibile sia dalla procedura che dal main
int n;                // variabile globale visibile sia dalla procedura che dal main

void stampaCubo()           // dichiarazione e definizione della procedura
{
    cub=x*x*x;
    cout<<"Il cubo di "<<x<<" vale "<<cub<<endl;
}

int main()
{
    cout<<"Inserisci un valore ";
    cin>>n;
    for (x=1;x<=n;x++)
        stampaCubo();       // chiamata della procedura
}
```

Variabili locali e globali

- La possibilità da parte del programma e delle varie procedure di vedere e modificare il valore di una variabile globale è un aspetto
  - Positivo: permetto lo scambio di informazioni
  - Negativo: la modifica del valore di una variabile globale da parte di una procedura potrebbe alterare il comportamento dell'intero programma (effetto collaterale)
- Esistono variabili che hanno significato solo all'interno di una procedura
- Queste variabili possono essere dichiarate all'interno della procedura (variabili locali) e quindi essere visibili sono in questa

Un esempio con variabili globali e locali

```
…
int x;                      // variabile globale visibile sia dalla procedura che dal main

void stampaCubo()           // dichiarazione e definizione della procedura
{
   int cub;                 // variabile locale visibile solo dalla procedura
   cub=x*x*x;
   cout<<"Il cubo di "<<x<<" vale "<<cub<<endl;
}

int main()
{
   int n;                     // variabile locale visibile solo nel main
   cout<<"Inserisci un valore ";
   cin>>n;
   for (x=1;x<=n;x++)
        stampaCubo();       // chiamata della procedura
}
```

Funzioni

- Anche per la funzione è valida l'analogia con la macchina. La Macchina-Funzione oltre ad eseguire il compito per il quale è stata costruita, restituisce il risultato.
- Anche in questo caso avremo una fase di dichiarazione-costruzione ed una di chiamata-avviamento.
- Nella fase di chiamata avremo la restituzione di un valore: **il risultato della funzione**.
- La funzione avrà un tipo (il tipo del valore restituito) e una terminazione esplicita (return seguito da una espressione che rappresenta il valore della funzione)

Esempio di funzione

```
int lancio()                // restituisce un valore casuale compreso fra 1 e 6
                            // (rappresenta il lancio di un dado)
{
    int faccia;                     // faccia del dado
    faccia = (rand() % 6) + 1;      // numero casuale compreso fra 1 e 6
    return faccia;
}

int main()
{
   srand(time(0));          // inizializzazione numeri casuali
   int numeroLanci;         // numero totale dei lanci da effettuare
   int l;                   // numero del lancio effettuato
   cout<<"Numero di lanci da effettuare: ";
   cin>>numeroLanci;
   for (l=1;l<=numeroLanci;l++)
         cout<<"Lancio N.ro "<<l<<" e' uscito il numero "<<lancio()<<endl;
}
```

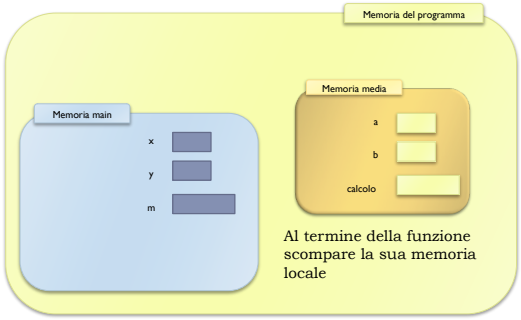
Scambio informazioni fra funzioni

- La memoria globale permette lo scambio di informazioni fra funzioni
- Risulta però un metodo
  - complesso (necessità di dichiarare tutte le variabili che servono a tutte le funzioni)
  - pericoloso (una funzione può alterare in modo inatteso una variabile globale)
- L'utilizzo dei parametri permette di ovviare al problema:
  - La funzione opera formalmente su variabili (parametri formali) che vengono associate a valori specifici al momento della chiamata (parametri attuali)

Parametri – passaggio per valore

```
…
float media(int a,int b)// calcola e restituisce la media fra i valori di a e di b
{
  // a e b sono parametri formali
  // al momento dell'esecuzione avviene l'assegnamento
  // fra il valore dei parametri attuali e i parametri formali
  float calcolo;
  calcolo=float(a+b)/2;
  return calcolo;
}
int main()
{
    int x,y;            // saranno utilizzati come parametri attuali
    float m;
    cout << "Inserisci due valori interi ";
    cin >> x;
    cin >> y;
    m = media(x,y);     // chiamata della funzione e passaggio dei parametri
    cout << "la media fra " << x << " e " << y << " e' "<< m << endl;
    m = media(x-1,3);   // il parametri attuale può essere una qualsiasi espressione (controllo sui tipi)
    cout << "la media fra " << x-1 << " e " << 3 << " e' "<< m << endl;
}
```

La memoria durante l'esecuzione



Passaggio per valore (non ha effetto)

- Il passaggio per valore non permette però di "ricordare" le modifiche apportate ai parametri da parte della funzione
- La memoria locale della funzione (che contiene i parametri formali) scompare nel momento in cui la funzione termina la sua esecuzione.

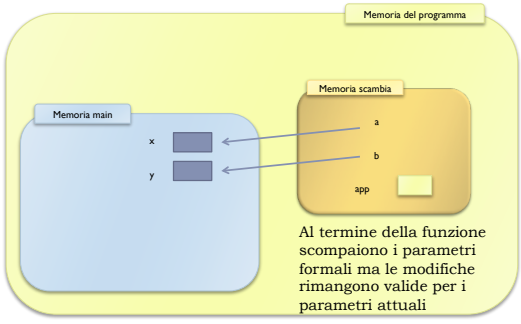
```
void scambia(int a,int b)// // scambia i valori di a e b
{
  int app;
  app = a;
  a = b;
  b = app;
}

int main()
{
    int x,y;            // saranno utilizzati come parametri attuali
    x = 3;
    y = 5;
    cout << "il valore di x e' " << x << " e il valore di y e' " << y << endl;
    scambia(x,y);
    cout << "il valore di x e' " << x << " e il valore di y e' " << y << endl;
}
```

Passaggio per riferimento

- Non viene più passato il valore ma un riferimento alla stessa variabile

```
void scambia(int &a,int &b)// // scambia i valori di a e b
{
  // a e b sono nuovi riferimenti ai parametri attuali
  // a "punta" alla stessa variabile x e b a y
  int app;
  app = a;
  a = b;
  b = app;
}
int main()
{
    int x,y;            // saranno utilizzati come parametri attuali
    x = 3;
    y = 5;
    cout << "il valore di x e' " << x << " e il valore di y e' " << y << endl;
    scambia(x,y);
    cout << "il valore di x e' " << x << " e il valore di y e' " << y << endl;
}
```

La memoria durante l'esecuzione



Elemento di array come parametro attuale

```
void stampaDivisori(int n)           // stampa i divisori del numero n
{
    int pDiv;       // possibile divisore
    cout << "Divisori di " << n << " :";
    for (pDiv=1; pDiv<=n; pDiv++)
        if ((n%pDiv)==0)
            cout << " " << pDiv;
    cout << endl;
}
int main()
{
    int valori[10];     // conterrà 10 valori casuali compre si fra 0 e 99
    srand(time(0));
    for (int i = 0; i<10 ; i++)
        valori[i]=rand()%100;
    for (int i = 0; i<10 ; i++)
        stampaDivisori(valori[i]);
}
```

Array come parametro

- E' necessario passare anche il numero di elementi

```
void stampaArray(int v[], int n)    // stampa gli elementi dell'array v di n elementi
{
    for (int i=0; i<n; i++)
        cout << v[i] << endl;
}


int main()
{
    int valori[10];     // conterrà 10 valori casuali compre si fra 0 e 99
    srand(time(0));
    for (int i = 0; i<10 ; i++)
        valori[i]=rand()%100;
    stampaArray(valori,10);                 // necessario passare anche la dimensione
}
```

Gli array sono sempre passati per riferimento

```
void stampaArray(int v[], int n)           // stampa gli elementi dell'array v di n elementi
{
    for (int i=0; i<n; i++)
        cout << v[i] << endl;
}
void raddoppiaValori(int v[], int n)        // raddoppia il valore degli elementi dell'array
{
    for (int i=0; i<n; i++)
        v[i] = v[i] * 2;
}
int main()
{
    int valori[10];     // conterrà 10 valori casuali compre si fra 0 e 99
    srand(time(0));
    for (int i = 0; i<10 ; i++)
        valori[i]=rand()%100;
    stampaArray(valori,10);                 // necessario passare anche la dimensione
    raddoppiaValori(valori,10);
    cout<<"Valori raddoppiati"<<endl;
    stampaArray(valori,10);
}
```

Procedure e funzioni. Perché?

- Evitare duplicazioni del codice
  - Con le procedure si evita di duplicare parti del codice sorgente, quando si chiama o invoca una procedura si esegue il codice corrispondente. A ogni nuova chiamata il suo codice è eseguito nuovamente.
  - La duplicazione pone due tipi di problemi:
    - Aumento della lunghezza del codice e quindi minore leggibilità
    - Difficoltà nell'apportare modifiche che devono essere effettuate in tutte le copie del codice
- … ma questa non è la motivazione più importante

Progettazione modulare

- Per affrontare problemi complessi si ricorre alla tecnica dei raffinamenti successivi che suggerisce di scomporre il problema in problemi più semplici (sottoproblemi)
- … e di applicare anche a questi sottoproblemi la stessa tecnica fino ad ottenere problemi facilmente risolvibili
- Questa tecnica è definita top-down:
- Si parte da una visione globale del problema (alto livello di astrazione) [top]
- Poi si scende nel dettaglio dei sottoproblemi diminuendo il livello di astrazione [down]
- Viene fornita inizialmente una soluzione del problema che non si basa però su operazioni elementari, ma sulla soluzione di sottoproblemi

Metodologia top down

- Se il sottoproblema è semplice allora viene risolto, viene cioè scritto l'algoritmo di risoluzione
- Se il sottoproblema è complesso viene riapplicato lo stesso procedimento scomponendolo in sottoproblemi più semplici
- Diminuisce il livello di astrazione (si affrontano problemi sempre più concreti)
- Diminuisce il livello di complessità (i sottoproblemi devono essere più semplici del problema che li ha originati)
- Fino ad arrivare alla stesura di tutti gli algoritmi necessari

Top Down e Bottom Up

- I modelli top-down e bottom-up (ing. dall'alto verso il basso e dal basso verso l'alto, rispettivamente) sono strategie di elaborazione dell'informazione e di gestione delle conoscenze, riguardanti principalmente il software …
- Nel modello **top-down** è formulata una visione generale del sistema senza scendere nel dettaglio di alcuna delle sue parti. Ogni parte del sistema è successivamente rifinita aggiungendo maggiori dettagli dalla progettazione.
- Nella progettazione **bottom-up** parti individuali del sistema sono specificate in dettaglio. Queste parti vengono poi connesse tra loro in modo da formare componenti più grandi, che vengono a loro volta interconnessi fino a realizzare un sistema completo.

Wikipedia