Informatica B — Prof. Martena, Morzenti — Tema B — 5 Febbraio 2004

Cognome:..... Nome:..... Matricola:.....

1 Funzioni: uso e passaggio dei parametri

Siano date le seguenti definizioni di tipo:

```
#define MAX 100

typedef struct {

   char nome[20];
   char cognome[20];
   int  salario;

} Impiegato;

typedef struct {

   int numImpiegati;
   Impiegato assunti[MAX];

} ImpiegatiAzienda;
```

Si scriva l'intestazione e l'implementazione delle seguenti funzioni:

1. Una funzione che prende in ingresso una variabile di tipo `ImpiegatiAzienda` ed un array di 2 interi in cui inserisce la posizione dei 2 impiegati con il salario più basso. Si assuma, per semplicità, che tutti gli impiegati abbiano salari differenti.

2. Una funzione che prende in ingresso una variabile di tipo `ImpiegatiAzienda` ed un array di 2 interi che contengono la posizione di 2 impiegati e modifica il salario di tali impiegati incrementandolo del 10 per cento. Per implementare quest'ultima funzione si usi una funzione (che si assume già implementata) con la seguente intestazione:

   ```
   void ModificaSalario(Impiegato * imp, int percentuale);
   ```

   che aggiorna di una determinata percentuale lo stipendio dell'impiegato passato come parametro.

N.B. Si faccia attenzione, nello scrivere le intestazioni delle funzioni, alla modalità di passaggio dei parametri da usare.

SOLUZIONE:

```
void calcolaImpiegatiPoveri(ImpiegatiAzienda azienda, int poveri[]){

  int i;

  /* metto in poveri[0] l'impiegato con il salario piu' basso ed in
     poveri[1] l'impiegato con il secondo salario piu' basso;
     all'inizio entrambi valgono 0, cioe' consideriamo il primo
     impiegato quello con il salario piu' basso. */

  poveri[0]=0;
  poveri[1]=0;

  for (i=1; i<azienda.numImpiegati; i=i+1){

    if (azienda.assunti[i].salario<azienda.assunti[poveri[0]].salario){
      poveri[1]=poveri[0];
      poveri[0]=i;
    } else if (azienda.assunti[i].salario<azienda.assunti[poveri[1]].salario){
      poveri[1]=i;
    }
  }
}

void anchePoveriRidono(ImpiegatiAzienda * azienda, int poveri[]){

  ModificaSalario(&azienda->assunti[poveri[0]], 10);
  ModificaSalario(&azienda->assunti[poveri[1]], 10);
}
```

2 Funzioni ricorsive

Sia data la seguente funzione ricorsiva:

```
int cheFa(int x){
  if (x==0){
    return 0;
  } else {
    return 3 + cheFa(x-1);
  }
}
```

1. Si indichi la sequenza delle invocazioni della funzione in corrispondenza dell'invocazione

   `cheFa(10);`

   riportando a fianco di ogni invocazione il risultato restituito dalla funzione;

2. Si indichi, in generale, qual è la funzione matematica calcolata dall'invocazione

   `cheFa(y);`

   quando l'argomento è un'espressione `y` avente valore intero positivo;

SOLUZIONE:

cheFa(10) = 30
cheFa(9) = 27
cheFa(8) = 24
cheFa(7) = 21
cheFa(6) = 18
cheFa(5) = 15
cheFa(4) = 12
cheFa(3) = 9
cheFa(2) = 6
cheFa(1) = 3
cheFa(0) = 0

cheFa(y) calcola il risultato della moltiplicazione di y per 3

3 Liste dinamiche

Sia data la seguente definizione di tipo per una lista di interi:

```
typedef struct _elem_lista {

        struct _elem_lista * next;
        int elem;

} ElemLista;
```

Sia dato il seguente prototipo di funzione:

```
int sostituisci(ElemLista * Lista, int elem, int newElem);
```

Tale funzione prende in ingresso il puntatore alla testa di una lista (`Lista`) e due interi (`elem` e `newElem`) e sostituisce tutte le occorrenze di `elem` in `Lista` con 3 nuovi elementi di tipo `ElemLista` collegati tra loro che contengono l'intero `newElem` ed infine restituisce il numero di elementi in più che la lista contiene rispetto a prima che venisse effettuata la sostituzione.

```
int sostituisci(ElemLista * Lista, int elem, int newElem){

  ElemLista *corr, *succ, *nuovo;
  int i,elementiAggiunti=0;

  corr=Lista;

  while(corr!=NULL){

    if (corr->elem==elem){

      /* se l'elemento attuale e' uguale a quello cercato lo
         sostituisco e poi memorizzo in succ il suo successore, che
         sara' l'elemento a cui dovro' ricongiungere i nuovi
         elementi che andro' a creare */

      corr->elem=newElem;
      succ=corr->next;

      /* Aggiungo gli elementi usando un ciclo for */

      for (i=0; i<2; i=i+1){

        nuovo=(ElemLista *)malloc(sizeof(ElemLista));
        nuovo->elem=newElem;
        corr->next=nuovo;
        corr=nuovo;

      }

      /* alla fine del ciclo in cui aggiungo gli elementi
         ricongiungo l'ultimo elemento creato con il successore che
         avevo identificato all'inizio del ciclo for ed incremento
         di due il numero di elementi aggiunti*/

      corr->next=succ;
      elementiAggiunti=elementiAggiunti + 2;
    }

    corr=corr->next;

  }

  return elementiAggiunti;

}
```

4 Sistemi Operativi

Nel contesto di un sistema operativo multitasking, si ipotizzi che il nucleo del sistema operativo tenga traccia dei processi pronti inserendoli in una struttura dati a lista collegata a puntatori, in cui in ogni elemento della lista siano memorizzati i dati relativi a uno dei processi correntemente pronti.
Si considerino i due seguenti, possibili modi di gestire la lista che memorizza l'insieme dei processi pronti in corrispondenza ai loro cambiamenti di stato.

1. Ogni processo che entra nello stato di pronto viene inserito all'inizio della lista (in prima posizione), mentre quando un processo deve essere fatto transitare dallo stato di pronto allo stato di in esecuzione si sceglie, tra quelli che si trovano nella lista dei processi pronti, quello che sta in ultima posizione;

2. Ogni processo che entra nello stato di pronto viene inserito all'inizio della lista (in prima posizione) e, quando un processo deve essere fatto transitare dallo stato di pronto allo stato di in esecuzione, si sceglie, tra quelli che si trovano nella lista dei processi pronti, quello che sta in prima posizione.

Indicare per quali delle due strategie (1) o (2) citate sopra sono verificate le seguenti due affermazioni.

1. I processi escono dallo stato di pronto nello stesso ordine in cui vi sono entrati. Vero per (1)? **SI**. Vero per (2)? **NO**.

2. I processi escono dallo stato di pronto in ordine inverso a quello in cui vi sono entrati. Vero per (1)? **NO**. Vero per (2)? **SI**.

Si consideri la seguente proprietà della gestione dei processi, considerata desiderabile per un sistema operativo multitasking: qualsiasi coppia di processi, che faccia uso della stessa quantit di risorse del calcolatore, a parità di tutti gli altri fattori impiega lo stesso tempo a terminare la propria esecuzione (chiamiamo questa proprietà **equità di trattamento dei processi**).

Una sola delle due strategie (1) e (2) di gestione dei processi pronti sopra descritte permette di assicurare l'equità di trattamento dei processi. Quale delle due? Motivare sinteticamente la risposta.

La strategia (1) garantisce l'equità di trattamento dei processi, perché, dal momento che i processi escono dallo stato di pronto nello stesso ordine in cui vi sono entrati, si evitano "sorpassi" tra processi pronti, cioè situazioni in cui un processo, passa dallo stato di pronto allo stato di in esecuzione prima di un altro processo che era entrato nello stato di pronto prima di lui.

Se si usa l'altra strategia, quale fenomeno si può verificare, nel caso più sfortunato, che viola gravemente l'equità di trattamento dei processi? Fornire una spiegazione sintetica.

Se si usa l'altra strategia, nel caso pi sfortunato un processo può rimanere indefinitamente nello stato di pronto senza mai fare alcun progresso, perché viene continuamente "superato" da altri processi entrati nello stato di pronto dopo di lui.