Fondamenti di Informatica T-1
modulo 2

Laboratorio 10:

preparazione alla prova d'esame

Esercizio 1

Esercizio 1 - Gestione degli impegni

- Gli ***impegni giornalieri*** dei dipendenti di un'azienda devono essere aggiornati con una serie di nuove richieste
- Gli ***impegni già fissati sono memorizzati in un file di testo***, all'interno del quale ogni riga rappresenta gli impegni di un dipendente, secondo il seguente formato:
  - Codice numerico del dipendente (intero)
  - Nome del dipendente (al più di 20 caratteri), seguito da un ';'
  - Sequenza di impegni del dipendente, costituita da
    - Ora di inizio dell'impegno, con formato HH:MM (HH e MM interi), seguita da uno spazio
    - Ora di fine dell'impegno, con formato HH:MM, seguita da un ';' (a meno che non si tratti dell'ultimo impegno, in questo caso c'è un fine linea)

Gestione degli impegni
Requisiti generali

- Si fissi come ***ipotesi*** che la ***sequenza di impegni sia ordinata rispetto al tempo***, e che ogni lavoratore abbia un massimo di 20 impegni. Nota: ogni impegno ha un orario di inizio superiore o uguale alle 09:00 e un orario di fine inferiore o uguale alle 18:00
- Un secondo ***file binario*** contiene le informazioni relative ad ***ulteriori richieste di impegni da eseguire (possibilmente) in giornata***. Il primo campo del file riporta il numero di richieste contenute, seguito dalle varie richieste. Ogni richiesta è costituita dal ***codice del dipendente*** a cui si riferisce e da un intero rappresentate i ***minuti da dedicare*** per l'impegno richiesto

Parte 1
Strutture dati e relative operazioni

- Definire delle strutture atte a contenere i dati relativi a dipendenti, impegni e riunioni
- Il tipo **event** deve contenere i dati di un impegno (orario di inizio e di fine) sfruttando una seconda struttura dati chiamata **time** (contenente ore e minuti)
- Il tipo **worker** contiene i dati relativi ad un dipendente: codice numerico, nome e sequenza di impegni da modellare come un vettore di **event *allocato staticamente***
  - *Nota: ricordarsi che servirà memorizzare anche la dimensione logica di tale vettore*
- Infine, il tipo **request** memorizza i dati di una richiesta di impegno, ovvero codice del dipendente e numero di minuti

Parte 1
Strutture dati e relative operazioni

Dichiarare e definire le seguenti funzioni:

- **timeDifference** deve prendere in ingresso due orari e restituirne la differenza in minuti (valore assoluto, ricordandosi di effettuare le opportune conversioni minuti/ore… la differenza tra 10:20 e 8:50 è pari a 90 minuti)
- **getTranslatedTime** deve prendere in ingresso un orario e un valore in minuti e restituire l'orario che si ottiene aggiungendo i minuti dati all'orario (ricordarsi di effettuare le opportune conversioni minuti/ore)

Parte 2
Lettura dai file

- Realizzare una funzione **readWorkers** che prenda in input il nome del file di testo contenente i dati dei dipendenti, lo apra in lettura e ***legga i dati di tutti i dipendenti memorizzandoli in una lista di worker***
- Realizzare una funzione **readRequests** che prenda in input il nome del file binario contenente i dati delle richieste, lo apra in lettura e legga i dati di tutte le richieste memorizzandoli in un ***vettore di meeting allocato dinamicamente della dimensione strettamente necessaria***. Si ricorda che il numero di richieste è memorizzato come primo campo del file
- Realizzare due funzioni di stampa verificando la corretta lettura dei dati

Parte 3
inserimento richiesta

Realizzare una funzione **insertNewEvent** che prenda in input una richiesta di impegno e la lista di dipendenti, ottenendo il dipendente coinvolto e cercando di inserire la richiesta nella sua agenda. Si adotti la seguente politica per l'inserimento:

- la richiesta deve essere inserita ***prima possibile tra i vari impegni già presi dal dipendente*** (ovvero appena il dipendente ha un tempo libero capace di contenere il nuovo impegno), senza ovviamente travalicare i limiti della giornata lavorativa (09:00 – 18:00); al fine di individuare l'ammontare dei tempi liberi, si utilizzi la funzione **timeDifference** sviluppata al punto 1.
- se è possibile inserire la richiesta in agenda, la funzione si deve incaricare di creare effettivamente il nuovo impegno (l'impegno partirà o alle 09:00 o al tempo di fine dell'impegno immediatamente precedente) e di inserirlo nel vettore degli impegni del dipendente sfruttando **l'inserimento ordinato**.
- La funzione deve restituire un booleano che attesti se l'inserimento è stato effettuato o meno

Parte 4
Gestione delle richieste

- Realizzare una funzione **handleRequests** che prenda in input una lista di dipendenti e un vettore di richieste, cercando di inserirle nelle varie agende. A tal fine, si scandisca il vettore di richieste, utilizzando la funzione **insertNewEvent** per l'inserimento.
  - Se la richiesta non è stata inserita, si stampi a video un messaggio di errore del tipo "il dipendente XXX non ha tempo sufficiente per gestire un impegno di YYY minuti"
- Realizzare una funzione **printWorkers** che stampi su un file di testo una lista di dipendenti e relativi impegni, con lo stesso formato utilizzato in lettura.
- Nel main, si utilizzino le funzioni sviluppate per acquisire i dati in input, gestire le richieste e stampare su file la lista dei dipendenti dopo l'aggiornamento.

Header funzionalità comuni (common.h/common.c)

```
#ifndef common
#define common
   #include <stdio.h>
   typedef enum{false, true} Boolean;
   int readField(char buffer[], char sep, FILE *f);
#endif
```

common.c implementa la readField come da lucidi.

Parte 1 – Strutture dati e funzionalità di agenda (agenda.h)

```
#ifndef agenda
#define agenda
   #define DIM_NAME 21
   #define MAX_EVENTS 20
   #include "common.h”

   typedef struct {
       int h;
       int m;
   } Time;
   typedef struct {
       Time startTime;
       Time endTime;
   } Event;
   …
```

Parte 1 – Strutture dati e funzionalità di agenda (agenda.h)

```
…
typedef struct {
    int code;
    char name[DIM_NAME];
    Event events[MAX_EVENTS];
    int nEvents;//dimensione logica di events
} Worker;
//manipolazione tempi (Es. 1)
int timeDifference(Time t1, Time t2);
Time getTranslatedTime(Time t, int displacement);
//scrittura delle strutture dati a video
void printTime(Time t);
void printEvent(Event e);
void printWorker(Worker w);
…
```

Parte 1 – Strutture dati e funzionalità di agenda (agenda.h)

```
…
//scrittura delle strutture dati su file (di testo)
void writeTime(FILE* f, Time t);
void writeEvent(FILE* f, Event e);
void writeWorker(FILE* f, Worker w);
//lettura delle strutture dati da file (di testo) (Es. 2)
Boolean readTime(FILE* f, Time* t);
Boolean readEvent(FILE* f, Event* e);
Boolean readWorker(FILE* f, Worker* w);
//funzioni per l'inserimento ordinato di eventi (Es. 4)
int compare(Event e1, Event e2);
Boolean insert(Event* events, Event el, int dim, int* elCount);
#endif
```

Parte 1 – Strutture dati e funzionalità di richiesta (request.h)

```
#ifndef req
#define req

   typedef struct {
        int workerCode;
        int minutes;
   } Request;

   //stampa a video di una richiesta e di un vettore di richieste
   void printRequest(Request r);
   void printRequests(Request* reqs, int dim);
   //lettura di un vettore di richieste da file binario (Es. 2)
   Request* readRequests(char* fileName, int* dim);

#endif
```

Parte 1 – Implementazione funzioni sui tempi (agenda.c)

```
#include "agenda.h"
#include "math.h"

int timeDifference(Time t1, Time t2)
{
   return abs(t1.m - t2.m + (t1.h - t2.h)*60);
}

Time getTranslatedTime(Time t, int displacement)
{
   Time t2;
   t2.m = (t.m + displacement) % 60;
   t2.h = t.h + (t.m + displacement) / 60;
   return t2;
}
```

Parte 2 – Lista di Workers (agenda.h)

```
#include "agenda.h”
typedef Worker Element;
```

Parte 2 – header gestione agende (agendaManagement.h)

```
#include "list.h"
#include "request.h”
//Stampa lista di workers a video
void printWorkers(List l);
//Lettura lista di workers da file di testo (Es. 2)
List readWorkers(char* fileName);
//Inserimento richiesta (Es. 3)
Boolean insertNewEvent(Request r, List workers);
// Gestione e stampa richieste (Es. 4)
void handleRequests(List workers, Request* reqs, int dimReqs);
void writeWorkers(char* fileName, List workers);
```

↑

Parte 2 – lettura Time e Event (agenda.c)

```
Boolean readTime(FILE *f, Time* t)
{
  return fscanf(f, "%d:%d", &t->h, &t->m) == 2;
}

Boolean readEvent(FILE *f, Event* e)
{
  if(! readTime(f, &e->startTime))
   return false;
  if(! readTime(f, &e->endTime))
   return false;
  else
   return true;
}
```

Parte 2 – lettura singolo Worker (agenda.c)

```
Boolean readWorker(FILE *f, Worker* w)
{
  int i = 0;
  Boolean ok;
  if(fscanf(f, "%d", &w->code) != 1)
   return false;
  if(readField(w->name, ';', f) != ';')
   return false;
  do {
   ok = readEvent(f, &(w->events[i]));
   if(ok) i++;
  }
  while(ok && fgetc(f) == ';');
  w->nEvents = i;
  return true;
}
```

Parte 2 – lettura Workers (agendaManagement.c)

```
List readWorkers(char* fileName)
{
  List workers = emptyList();
  Worker w;
  FILE* f = fopen(fileName, "r");
  if(f == NULL)
   abort();
  while(readWorker(f, &w))
   workers = cons(w, workers);
  fclose(f);
  return workers;
}
```

Parte 2 – Stampa Time e Event (agenda.c)

```
void printTime(Time t)
{
  printf("%02d:%02d",t.h,t.m);
}

void printEvent(Event e)
{
  printTime(e.startTime);
  printf("-");
  printTime(e.endTime);
}
```

Parte 2 – Stampa singolo Worker (agenda.c)

```
void printWorker(Worker w)
{
  int i;
  printf("%d) %s", w.code, w.name);
  printf("\n\t");
  for(i = 0; i < w.nEvents; i++)
  {
   printf(" ");
   printEvent(w.events[i]);
  }
}
```

Parte 2 – Stampa Workers (agendaManagement.c)

```
void printWorkers(List l)
{
  while(!empty(l))
  {
   printWorker(head(l));
   printf("\n");
   l = tail(l);
  }
}
```

Parte 2 – Lettura richieste (request.c)

```
Request* readRequests(char* fileName, int* dim)
{
  FILE* f = fopen(fileName, "rb");
  Request* reqs;
  fread(dim, sizeof(int), 1, f);
  reqs = (Request *) malloc(*dim * sizeof(Request));
  fread(reqs, sizeof(Request), *dim, f);
  return reqs;
}
```

Parte 2 – Stampa richieste (request.c)

```
void printRequest(Request r) {
  printf("Worker: %d, amount: %d",
                          r.workerCode, r.minutes);
}

void printRequests(Request* reqs, int dim) {
  int i;
  for(i = 0; i < dim; i++)
  {
   printRequest(reqs[i]);
   printf("\n");
  }
}
```

Parte 3 – Comparazione tra eventi (agenda.c)

```
//Comparazione tra eventi (per l'inserimento
  ordinato). La comparazione è effettuata sui
  tempi di inizio degli eventi
int compare(Event e1, Event e2)
{
  int comp = e1.startTime.h - e2.startTime.h;
  if(comp == 0)
   return e1.startTime.m - e2.startTime.m;
  else
   return comp;
}
```

Parte 3 – Inserimento ordinato di eventi (agenda.c)

```
Boolean insert(Event* events, Event el, int dim, int* elCount)
{
   int i = 0, j;
   Boolean found = false;
   if(*elCount < dim)
   {
            while(i < *elCount && !found)
            {
                     if(compare(el, events[i]) < 0)
                              found = true;
                     else
                              i++;
            }
            if(found)
                     for(j = *elCount; j >= i; j--)
                              events[j] = events[j-1];
                     events[i] = el;
                     (*elCount)++;
                     return true;
   }
   return false;
}
```

Parte 3 – Inserimento richiesta (agendaManagement.c)

```
Boolean insertNewEvent(Request r, List workers)
{
  int i = 0;
  Boolean slotFound = false;
  int freeTime;
  Time tEndPrec;
  Time tStartNext;
  Worker* w;
  tEndPrec.h = 9;
  tEndPrec.m = 0; // si parte dalle 9:00
  w = findWorker(r.workerCode, workers);
  if(w == NULL) //dipendente non trovato
   return false;
  …
```

Parte 3 – Inserimento richiesta (agendaManagement.c)

```
…
while(i <= w->nEvents && !slotFound)
{
  //l'ultimo controllo considera le 18:00
  if(i == w->nEvents)
  {
   tStartNext.h = 18;
   tStartNext.m = 0;
  }
  else
   tStartNext = w->events[i].startTime;
  freeTime = timeDifference(tStartNext, tEndPrec);
  slotFound = freeTime >= r.minutes;
  …
```

Parte 3 – Inserimento richiesta (agendaManagement.c)

```
    …
    if(slotFound)
    {
      Event e;
      e.startTime = tEndPrec;
      e.endTime = getTranslatedTime(tEndPrec, r.minutes);
      insert(w->events, e, MAX_EVENTS, &w->nEvents);
    }
    if(i < w->nEvents)
      tEndPrec = w->events[i].endTime;
    i++;
  }
  return slotFound;
}
```

Parte 4 – Gestione richieste (agendaManagement.c)

```
void handleRequests(List workers,
                                Request* reqs, int dimReqs)
{
  Boolean ok;
  int i;
  for(i = 0; i < dimReqs; i++)
  {
   ok = insertNewEvent(reqs[i], workers);
   if(!ok)
     printf(   "Il dipendente %d non ha tempo
                 sufficiente per gestire un impegno di
          %d minuti\n",
               reqs[i].workerCode, reqs[i].minutes);
  }
}
```

Parte 4 – Stampa Time e Event su file (agenda.c)

```
void writeTime(FILE* f, Time t)
{
  fprintf(f, "%02d:%02d", t.h, t.m);
}

void writeEvent(FILE* f, Event e)
{
  writeTime(f, e.startTime);
  fputc(' ',f);
  writeTime(f, e.endTime);
}
```

Parte 4 – Stampa singolo Worker su file (agenda.c)

```
void writeWorker(FILE* f, Worker w)
{
  int i;
  fprintf(f, "%d%s", w.code, w.name);
  for(i = 0; i < w.nEvents; i++)
  {
   fputc(';', f);
   writeEvent(f, w.events[i]);
  }
}
```

Parte 4 – Stampa Workers su file (agendaManagement.c)

```
//La stampa rispetta l'ordine del file di lettura!
void writeWorkersRecursive(FILE* f, List workers)
{
  if(!empty(workers))
  {
   writeWorkersRecursive(f, tail(workers));
   writeWorker(f, head(workers));
   fputc('\n', f);
  }
}

void writeWorkers(char* fileName, List workers)
{
  FILE* f = fopen(fileName, "w");
  writeWorkersRecursive(f, workers);
}
```

Esercizio 2

Esercizio 2: Prenotazione di pizze

- Una catena americana di pizzerie permette la prenotazione di pizze tramite il proprio sito Web
- Si deve realizzare un programma che, forniti un insieme di ordini di pizze e il listino dei prezzi dei singoli ingredienti, calcoli la spesa da fatturare ad ogni cliente. Le informazioni sono contenute in due differenti file, **ingredienti.txt** e **ordini.bin**.
- Il file di testo **ingredienti.txt** memorizza, in ogni riga,
  - nome dell'ingrediente (al più 24 caratteri, senza spazi)
  - prezzo di tale ingrediente (un float)

Prenotazione di pizze
Requisiti generali

- Il file binario **ordini.bin** memorizza invece gli ordini:
  - all'inizio è salvato il numero di ordini presenti in tutto il file (int)
  - Di seguito, sono registrate ***tutte le pizze ordinate***: esattamente in questo ordine, per ogni pizza, sono salvati
    - nome del cliente che ha ordinato la pizza (al più 64 caratteri)
    - numero di ingredienti presenti su quella pizza (al più 10)
    - array di 10 elementi (non tutti usati) di ingredienti (composti dalla coppia nomeIngrediente-prezzo)
- Ovviamente il file ***può contenere più pizze ordinate dalla stessa persona*** e le liste degli ingredienti ivi registrati, anche se composte dalla coppia nomeIngrediente-prezzo, in realtà riportano in modo uniforme solo il nome dell'ingrediente: infatti il prezzo è soggetto a frequenti modifiche (fa testo in tal senso il solo listino prezzi ufficiale dato nel file precedente)

Parte 1
Lettura e scrittura degli ingredienti

- Il candidato realizzi un modulo per la ***gestione del listino degli ingredienti***
- In particolare, definisca una opportuna struttura dati **ingrediente**, per tenere traccia del nome di un ingrediente (al più 24 caratteri senza spazi) e del suo prezzo (double)
- Il candidato poi definisca:
  - insieme di primitive opportune per poter usare/gestire liste di strutture dati **ingrediente** (si faccia riferimento, a tal scopo, alle primitive sulle liste viste a lezione)
  - funzione **leggiIngredienti(…)** che, ricevuto in ingresso il nome di un file, apra il file e restituisca una lista di strutture dati **ingrediente** lette da tale file
  - funzione **scriviIngredienti(…)** che, ricevuto come parametri d'ingresso il nome di un file di testo e una lista di strutture dati **ingrediente**, scriva le informazioni relative agli ingredienti sul file indicato. In particolare, su ogni riga si scriva il nome dell'ingrediente e, separato da uno spazio, il suo prezzo
- Nel main(), testare il funzionamento del modulo utilizzando il file di testo dato

Parte 2
Modifica dei prezzi

- Il candidato estenda il modulo definito al punto precedente realizzando:
  - funzione **trovaPrezzo(…)** che, ricevuti in ingresso il nome di un ingrediente e una lista di strutture dati **ingrediente**, restituisca il prezzo di tale ingrediente
  - funzione **aggiornaPrezzo(…)** che, ricevuti in ingresso la lista di strutture dati **ingrediente**, il nome di un ingrediente e un nuovo prezzo, restituisca la lista di ingredienti dove il prezzo dell'ingrediente specificato è stato aggiornato al nuovo valore. A tal scopo, il candidato consideri di accedere alla lista usando la notazione a puntatori
- Nel main() il candidato modifichi la lista di ingredienti letti nell'esercizio precedente e controlli, salvando la lista aggiornata su un file, che tale modifica sia avvenuta con successo

Parte 3
Gestione delle pizze

- Il candidato realizzi un modulo per la ***gestione di una singola pizza***
- In particolare, si definisca una struttura dati **Pizza** contenente le seguenti informazioni (esattamente in questo ordine):
  - nome del cliente che ha ordinato la pizza (al più 64 caratteri)
  - numero di ingredienti (int)
  - array di strutture dati **ingrediente**, di dimensione statica di 10 elementi
- Tipicamente una pizza conterrà meno di 10 ingredienti: il numero di tali ingredienti è indicato nell'apposito campo
- Il candidato poi implementi una funzione **calcolaPrezzo(…)** che, ricevuti in ingresso una struttura dati di tipo **Pizza** e una lista di strutture dati **ingrediente**, calcoli il costo di tale pizza e lo restituisca come risultato della funzione

Parte 4
Gestione degli ordini

- Il candidato realizzi un modulo di gestione degli ordini, che sono registrati su un file binario (a titolo di esempio, si consideri il file **ordini.bin** dato)
- Su tale file viene salvato innanzitutto il numero di pizze ordinate (int) e a seguire delle strutture dati di tipo **Pizza**
- Uno stesso cliente può ordinare più pizze, che possono essere registrate nel file anche in ordine non consecutivo; il file può contenere gli ordini relativi a diversi clienti. Il candidato implementi:
  - funzione **leggiOrdini(…)** che, ricevuto in ingresso il nome di un file di ordini, apra tale file, legga il numero di pizze ivi registrate, allochi dinamicamente memoria sufficiente, e legga le strutture dati di tipo **Pizza** presenti in tale file. La funzione dovrà restituire un ***puntatore all'area di memoria allocata*** e, tramite un parametro passato per riferimento, il numero di pizze lette
  - funzione **sort(…)** che, utilizzando un algoritmo di ordinamento a scelta del candidato (tra quelli visti a lezione), ordini un array di strutture dati di tipo **Pizza** in base al nome del cliente che ha ordinato tale pizza

Parte 5
Calcolo della spesa dei clienti

- Il candidato, utilizzando le funzioni definite negli esercizi precedenti, provveda a leggere il listino degli ingredienti dal file **ingredienti.txt**, aggiorni il prezzo dell'ingrediente "salame_piccante" a 2.00 euro e salvi (sovrascrivendo il file originale) la nuova lista di prezzi nello stesso file
- Quindi il candidato legga dal file **ordini.bin** un array di pizze richieste da diversi clienti, ordini tale array in base al nome del cliente e calcoli per ogni cliente la spesa totale (data dalla somma del costo di ogni pizza richiesta dallo stesso cliente)

  Suggerimento: si sfrutti il fatto che l'array è ordinato proprio in base al nome del cliente e che, quindi, pizze richieste dalla stessa persona risulteranno essere adiacenti

Parte 1 – Strutture dati e funzionalità ingredienti (listino.h)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifndef LISTINO
#define LISTINO
#define MAX 25
   typedef struct ingrediente
   {
        char nome[MAX];
        float prezzo;
   } Ingrediente;
   //lista di ingredienti
   typedef struct item_struct
   {
        Ingrediente value;
        struct item_struct *next;
   } item;
   typedef item *list;
```

Parte 1 – Strutture dati e funzionalità ingredienti (listino.h)

```
   //Lettura ingredienti (parte 1)
   list leggiIngredienti(char *filename);
   //Scrittura ingredienti (parte 1)
   int scriviIngredienti(char *filename, list l);
   //Ricerca prezzo (parte 2)
   float trovaPrezzo(char *nome, list l);
   //Aggiornamento prezzo (parte 2)
   list aggiornaPrezzo(list l, char *nome, float nuovoPrezzo);

   //Funzioni ADT lista (di ingredienti)
   list emptyList();
   int isEmpty(list l);
   list cons(Ingrediente newValue, list l);
   Ingrediente head(list l);
   list tail(list l);
#endif
```

Parte 1 – Lettura ingredienti (listino.c)

- Nota: per l'implementazione delle funzionalità relative alla lista si consultino le slide

```
list leggiIngredienti(char *filename)
{
   FILE * fp;
   char nome[MAX];
   float prezzo;
   Ingrediente temp;
   list result = emptyList();
   if ((fp = fopen(filename, "r")) == NULL)
        return result;
   while (fscanf(fp, "%s %f", nome, &prezzo) == 2)
   {
        strcpy(temp.nome, nome);
        temp.prezzo = prezzo;
        result = cons(temp, result);
   }
   fclose(fp);
   return result;
}
```

Parte 1 – Scrittura ingredienti (listino.c)

```
int scriviIngredienti(char *filename, list l)
{
   FILE * fp;
   if ((fp = fopen(filename, "w")) == NULL)
   {
        return -1;
   }
   while (!isEmpty(l))
   {
        fprintf(fp, "%s %f\n", head(l).nome, head(l).prezzo);
        l = tail(l);
   }
   fclose(fp);
   return 0;
}
```

Parte 2 – Ricerca prezzo (listino.c)

```
float trovaPrezzo(char *nome, list l)
{
   while (!isEmpty(l))
   {
        if (strcmp(head(l).nome, nome) == 0)
        {
               return head(l).prezzo;
        }
        else
        {
               l = tail(l);
        }
   }
   return -1;
}
```

Parte 2 – Aggiornamento prezzo (listino.c)

```
list aggiornaPrezzo(list l, char *nome, float nuovoPrezzo)
{
   list temp = l;
   int trovato = 0;
   while (!isEmpty(temp) && !trovato)
   {
        if (strcmp(head(temp).nome, nome) == 0)
        {
                trovato = 1;
                temp->value.prezzo = nuovoPrezzo;
        }
        else
        {
                temp = tail(temp);
        }
   }
   return l;
}
```

Parte 3 – Strutture dati e funzionalità pizze (pizza.h)

```
#ifndef PIZZA
#define PIZZA

   #include <stdio.h>
   #include <string.h>
   #include "listino.h”
   typedef struct pizza
   {
        char nomeCliente[65];
        int numeroIngredienti;
        Ingrediente topping[10];
   } Pizza;

   float calcolaPrezzo(Pizza p, list l); //Parte 3
   Pizza leggiPizza(FILE * fp); //Supporto in parte 4

#endif
```

Parte 3 – Calcolo prezzo (pizza.c)

```
float calcolaPrezzo(Pizza p, list l)
{
   int i;
   float result = 0;
   float temp;
   for (i = 0; i < p.numeroIngredienti; i++)
   {
        temp = trovaPrezzo(p.topping[i].nome, l);
        if (temp < 0)
        {
               return -1;
        }
        result = result + temp;
   }
   return result;
}
```

Parte 4 – Funzionalità ordini (ordini.h)

```
#ifndef ORDINI
#define ORDINI

   #include <stdio.h>
   #include <string.h>
   #include "pizza.h”

   Pizza * leggiOrdini(char *filename, int *numPizze);
   void naiveSortR(Pizza a[], int dim);

#endif
```

Parte 4 – Lettura singola pizza da file binario (pizza.c)

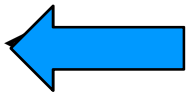
```
Pizza leggiPizza(FILE *fp)
{
   Pizza result;
   if (!feof(fp))
   {
        fread(&result, sizeof(Pizza), 1, fp);
   }
   return result;
}
```

Parte 4 – Lettura ordini (ordini.c)

```
Pizza* leggiOrdini(char *filename, int *numPizze)
{
   FILE *fp;
   Pizza *result;
   int temp;
   *numPizze = 0;
   if ((fp=fopen(filename, "rb")) == NULL)
        return NULL;
   fread(&temp, sizeof(int), 1, fp);
   result = (Pizza *) malloc(sizeof(Pizza) * temp);
   while (*numPizze < temp)
   {
        result[*numPizze] = leggiPizza(fp);
        *numPizze = *numPizze + 1;
   }
   fclose(fp);
   return result;
}
```

Parte 4 – Ordinamento ordini con naïve sort (ordini.c)

```
void naiveSortR(Pizza a[], int dim)
{
   int i, posmin;
   Pizza min;
   if (dim == 1)
        return;
   for (posmin = 0, min = a[0], i = 1; i < dim; i++)
   {
        if (strcmp(a[i].nomeCliente,min.nomeCliente)
        {
                posmin = I;
                min = a[i];
        }
   }
   if (posmin != 0)
        swap(&a[0], &a[posmin]);
   naiveSortR(&a[1], dim - 1);
}
```



```
void swap(Pizza *a, Pizza *b)
{
  Pizza tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Parte 5 – Calcolo spesa clienti (main.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "listino.h”
#include "pizza.h”
#include "ordini.h”

int main(void)
{
   list l;
   Pizza *p;
   int numPizze;
   int i;
   char *nome;
   float totale;
   l = leggiIngredienti("ingredienti.txt");
   l = aggiornaPrezzo(l, "salame_piccante", 2.00);
   scriviIngredienti("ingredienti.txt", l);
   …
```

Parte 5 – Calcolo spesa clienti (main.c)

```
…
p = leggiOrdini("ordini.bin", &numPizze);
naiveSortR(p, numPizze);
for (i=0; i<numPizze; )
{
     totale = calcolaPrezzo(p[i], 1);
     nome = p[i].nomeCliente;
     i++;
     while (strcmp(nome, p[i].nomeCliente) == 0)
     {
            totale = totale + calcolaPrezzo(p[i], 1);
            i++;
     }
     printf("%s deve pagare %f euro\n", nome, totale);
}
system("PAUSE");
return (0);
}
```