```
/*
  Vous developpez un logiciel de simulation de gestion de stock pour un magasin de chauss
ure.

  Chaque chaussure vendu dans ce magasin existe sous 4 marques
      (nike,addidas,puma,carrefour), et est disponible sous 3 pointures (38,40 et 42).
  Une chaussure d'un type donne (marque+pointure) possede un prix, et le magasin stocke
      le nombre restant de chaussure de ce type.


  Un client qui possede une certaine somme d'argent peut acheter une chaussure si
      le magasin la possede encore en stock et si celui-ci possede suffisamment d'argent.

  Vous devez implementer la fonction d'achat d'une chaussure par un client donne.


  Question:

  1- Completez le contrat de la fonction [client_achete_chaussure] d'apres
      une methodologie par contrat.
  2- Ecrivez le code de la fonction qui implemente le contrat que vous avez defini.

  Note: Le contrat que vous definissez vous est propre. Il n'existe pas de solution
      unique, definissez le de maniere logique vis a vis d'un tel logiciel.

 */


//Inclusions
#include <stdio.h>
#include <assert.h>


#define NOMBRE_DE_TYPE 4 //le nombre de marque de chaussure
//les differentes marques
enum marque_de_chaussure {nike,addidas,puma,carrefour};

#define NOMBRE_DE_POINTURE 3 //le nombre de pointures differentes
//les differentes pointures
enum pointure_de_chaussure {pointure_38,pointure_40,pointure_42};

struct stock_chaussure //l'enregistrement d'un type de chaussure specifique
{
  int prix;            //le prix de cette chaussure
  int nombre_en_stock; //le nombre restant disponible en magasin
};

struct stock_magasin //le stock total de chaussures du magasin
{
  // le stock classe les differents types de chaussures: marques/pointures sous forme de
tableau
  struct stock_chaussure ensemble_chaussure[NOMBRE_DE_TYPE][NOMBRE_DE_POINTURE];
};


#define NOMBRE_MAX_LETTRE 50
struct client // enregistrement d'un client
{
  char nom[NOMBRE_MAX_LETTRE]; //son nom
  int monnaie;                 //la quantitee d'argent qu'il possede
};

// Fonction d'initialisation du stock du magasin (implementation non visible)
void stock_magasin_initialise(struct stock_magasin* stock);

// Fonction d'achat d'une chaussure par un client (contrat a completer, implementation a
donner)
void client_achete_chaussure(struct client* client_courant,
                             struct stock_magasin* stock,
                             enum marque_de_chaussure type_souhaite,
                             enum pointure_de_chaussure pointure_souhaite);
```

```
//un exemple d'appel de ces fonctions
int main()
{
  struct stock_magasin stock;
  stock_magasin_initialise(&stock);

  struct client client_1={"Charles",300};
  struct client client_2={"Raymond",20};
  struct client client_3={"Bernard",150};

  client_achete_chaussure(&client_1,&stock,nike,pointure_38);
  client_achete_chaussure(&client_2,&stock,addidas,pointure_42);
  client_achete_chaussure(&client_3,&stock,carrefour,pointure_40);
  client_achete_chaussure(&client_3,&stock,nike,pointure_40);

  return 0;
}


//exemple de fonction d'initialisation
void stock_magasin_initialise(struct stock_magasin* stock)
{
  unsigned int k_type=0;
  unsigned int k_pointure=0;
  for(k_type=0;k_type<NOMBRE_DE_TYPE;k_type++)
    for(k_pointure=0;k_pointure<NOMBRE_DE_POINTURE;k_pointure++)
      {
        stock→ensemble_chaussure[k_type][k_pointure].prix=60;
        stock→ensemble_chaussure[k_type][k_pointure].nombre_en_stock=3;
      }
}
```

```
//*********************************************************//
//   FONCTION client_achete_chaussure                      //
//*********************************************************//


// Fonction d'achat d'une chaussure:
//  Un client peut acheter la chaussure de son choix si il possede l'argent suffisant et
si la chaussure et en stock.
//
// Necessite en entree:
// - Un pointeur (non constant) non NULL vers un client.
// - Un pointeur (non constant) non NULL vers un stock de magasin.
// - Une marque de chaussure souhaitee par le client comprise dans les marques disponible
s (4 marques dispo)
// - Une pointure de chaussure souhaitee par le client comprise dans les pointures dispon
ibles (3 pointures dispo)
//
// Garantie en sortie:
// - Si la chaussure n'est pas disponible en stock, alors on affiche a l'ecran: "Plus de
chaussure pour [NOM]". Le stock n'est pas modifie, et le client garde son argent.
// - Sinon, si la chaussure coute plus chere que l'argent possede par le client, alors on
 affiche a l'ecran: "Chaussure trop chere pour [NOM]". Le stock n'est pas modifie, et le
client garde l'argent qu'il avait.
// - Sinon, si la chaussure est dispo et que l'argent du client est suffisant, alors
//        - le nombre de chaussure du stock (pointee par le client) est decrementee de 1
//        - l'argent du client est decrementee du prix de la chaussure.
```

```
void client_achete_chaussure(struct client* client_courant,
                             struct stock_magasin* stock,
                             enum marque_de_chaussure type_souhaite,
                             enum pointure_de_chaussure pointure_souhaite)
{

  //Verification du contrat en entree:
  assert(type_souhaite<NOMBRE_DE_TYPE);
  assert(pointure_souhaite<NOMBRE_DE_POINTURE);
  assert(client_courant≠NULL);
  assert(stock≠NULL);
  //note 1: l'aspect non constant des pointeurs est verifiee par le compilateur.
  //note 2: les enum etant des entiers non signes, ils sont forcement positifs (verifiee
par le compilateur)

  //recuperation des informations sur la chaussure voulue.
  struct stock_chaussure* chaussure_voulue= &stock→ensemble_chaussure[type_souhaite][poi
nture_souhaite];

  //verification utilisateur:
  // - nombre de chaussure en stock > 0
  // - argent du client > prix de la chaussure.
  if(chaussure_voulue→nombre_en_stock≤0)
    {
      printf("Plus de chaussure pour %s\n",client_courant→nom);
      return ;
    }
  if(client_courant→monnaie<chaussure_voulue→prix)
    {
      printf("Chaussure trop chere pour %s\n",client_courant→nom);
      return ;
    }

  //programmation defensive juste au cas ou on se trompe dans les if.
  //(pas indispensable, mais il vaut mieux verifier trop de fois que pas assez)
  assert(client_courant→monnaie≥chaussure_voulue→prix) ;
  assert(chaussure_voulue→nombre_en_stock>0);


  //achat de la chaussure:
  //  - decrementation du nombre de chaussure en stock
  //  - decrementation de l'argent du client
  chaussure_voulue→nombre_en_stock--;
  client_courant→monnaie-=chaussure_voulue→prix;
}
```