Exercice 1:

```
if (x)
{
    y=x;
}
else
{
    y=x;
}
if (y)
{
    y=x;
}
else
{
    y=x;
}
```

Exercice 2
A,B,C boolean
if (A&&B) then
  return X+Y
else if (C) then
   return Y
else
   return X
fsi
Trouver les jeux d'essais minimum pour avoir une couverture instruction, puis une couverture branche.
A,B,C boolean
if (A&B) then
  Z = X+Y
End if
if (C) then
   z = z + X
fsi
return z
Trouver les jeux d'essais minimum pour avoir une couverture instruction, puis une couverture branche.

TP

Créer les tests qui permettront de valider :

la methode computePrice(age,tourist) de la classe ratp

règle de gestion : if age <= 12 le prix du ticket (1,5) est divisé par deux.
Si touriste le prix est multiplié par 2/

INSTRUCTIONS

1/ créer un premier test avec

assertEquals : `assert_float_equal (float  expected, float actual, float epsilon)`

forcer l'échec,

Q1: Pourquoi a-t-on besoin d'un epsilon

Réponse : Parce que la précision des flottants n'est pas exact, l'epsilon est la distance entre deux réels en dessous duquel on considère qu'ils sont égaux.

2/ créer un second test et utiliser l'assertion
assertTrue : assert_true `(boolean condition)`

forcer l'échec

Q2: que vaut-il mieux utiliser comme type d'assertion ?

Réponse : Si on veut tester spécifiquement l'égalité de deux valeurs numériques avec une précision donnée, assertEquals avec un epsilon de précision est le meilleur choix. Par contre, si on veut tester une condition générale qui doit être vraie, alors assert_true est plus approprié.

Modifier les tests pour utiliser les setup et teardown in main.c
static int setup(void **state) { (void) state; printf("setUp"); return 0; }
static int teardown(void **state) { (void) state; printf("tearDown"); return 0; }

4/ Finaliser les tests en utilisant les techniques de test.

Q4: pourquoi est-il préférable de faire un test pour chaque cas plutôt qu'un test avec plusieurs assertions?
Pour l'isolation des tests ce qui rend les tests plus clairs et plus faciles à comprendre.
Pour identifier les erreurs et déterminer rapidement quelle partie précise et simple du code a échoué.

Pour la maintenabilité, pour que les modifications soient plus localisées et moins susceptibles d'entraîner des effets indésirables.

5/ Implémenter la couverture → modifier le makefile :

```
CXX = gcc
LDFLAGS =   --coverage
OPTION = -Wall -fexceptions  -fprofile-arcs -ftest-coverage
INCS = -I.  -I"."
OBJS = $(SRC:.c=.o)

SRC = ratp.c  main.c cmocka.c

all: $(OBJS)
    $(CXX)  $(LDFLAGS) $(INCS) -o testAll  $(OBJS)
exec : all
    testAll.exe
%.o: %.c
    $(CXX) $(OPTION) -c  $< -o $@ $(INCS)
clean:
    rm $(OBJS)
mrproper: clean
    rm testAll.exe
    rm result.xml
```

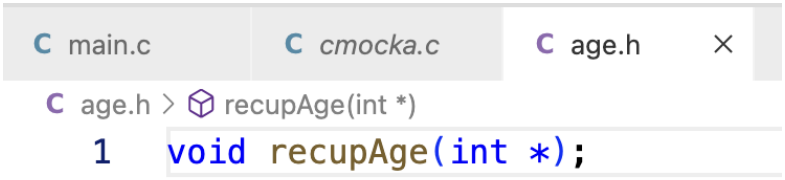
Ou LDFLAGS= -lgcov --coverage (depend des plateformes)
Exécuter les tests
Puis la commande gcov -b -c ratp.c → elle génére un fichier résultat ratp.c.gcov
Pip install gcovr → installe un module python qui permet de mieux visualiser la couverture.

gcovr --html-details result.html → pour avoir le résultat formatté.

6/ Mock
→ modifier le code ratp tel que :

```c
{
    int limitAge;
    int *plimit = &limitAge;
    recupAge(plimit);
    if (age>limitAge)
        return Price;
    else
        return Price/2;
}
```

Et créer le .h suivant :



Créer le mock de cette fonction dans le test et modifier le test en conséquence.

7/ mock : vérifier que le mock a bien été appelé une seule fois.
Avec le fichier html, on voit que la mock a bien été utilisé,