Solution exercice 2.6

1) OK.
   pi1 désigne le même entier que pi2 (soit, dans notre cas, 2)

2) Produit un warning : `assignment from incompatible pointer type`
   Il est par contre possible d'écrire :
   pd = (double*) pi1;

3) Produit un warning : `assignment from incompatible pointer type`.
   Il est par contre possible d'écrire :
   pi1 = (int*) pd;
   A noter que dans les cas 2 et 3), les conversions explicites peuvent conduire ensuite, lors du déréférencement, à des résultats non escomptés.
   Exemple :
   printf("%d\n", *pi1);
   n'affichera pas 3 comme (peut-être) espéré.

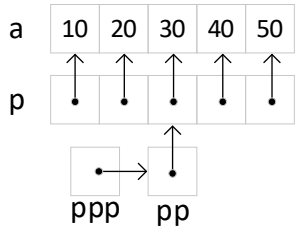
4) OK.
   pv contient maintenant l'adresse contenue dans pi1
   … mais attention : cette adresse ne peut pas être exploitée sans autres.
   Si, après avoir effectué l'affectation pv = pi1, on tente d'écrire, par exemple :
   printf("%d\n", *pv);
   , on obtient une erreur à la compilation.
   On peut par contre écrire :
   printf("%d\n", *((int*)pv));

5) Idem que 4)

6) OK.
   En C (mais pas en C++), un pointeur de type void* peut être converti implicitement en un pointeur d'un autre type

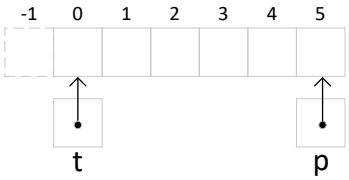
7) OK.
   Affecte à pi1 l'adresse contenue dans pi2 et la condition est vraie si pi2 n'est pas NULL

Solution exercice 2.7



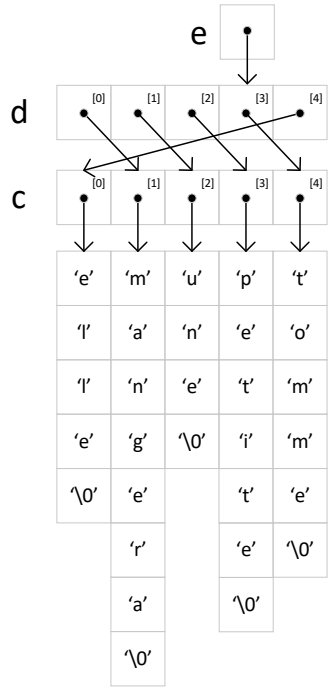

```
1) 10
2) 10
3) 30
4) 40
5) 20
6) 11
7) 20
8) 1
```

Solution exercice 2.8




```
0 1 2 3 4 5
0 0 1 1 2 2
```

Solution exercice 2.9




```
a) c[3][0]              => 'p'
b) (**d)[5]             => 'r'
c) (**e)[*d-c]          => 'o'
d) (d[3]-3)[0][3]       => 'g'
e) **d + 5              => "ra"
f) *d[3] + 2            => "mme"
g) *(*e[-3] + 5)        => 'r'
h) **c                  => 'e'
i) e[0][0][e-d]+1       => code ASCII de 'n' // car 'm' + 1 = 'n'
j) 0[c][0] - 'd' + 'B' => code ASCII de 'C' // car tab[i] = i[tab]
```

Solution exercice 2.12

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void inverser(int* debut, int* fin);
void afficher(const int tab[], size_t taille);
void test(int tab[], size_t taille);

int main(void) {

   {
      int tab[] = {1};
      test(tab, sizeof(tab) / sizeof(int));
   }

   {
      int tab[] = {1, 2};
      test(tab, sizeof(tab) / sizeof(int));
   }

   {
      test((int[]){1, 2, 3}, 3); // Autre manière de procéder
   }

   return EXIT_SUCCESS;
}

void inverser(int* debut, int* fin) {
   assert(debut != NULL);
   assert(fin != NULL);
   while (debut < fin) {
      int tampon = *debut;
      *debut++ = *fin;
      *fin-- = tampon;
   }
}

void afficher(const int tab[], size_t taille) {
   assert(tab != NULL);
   printf("[");
   for (size_t i = 0; i < taille; ++i) {
      if (i > 0)
         printf("%s", ", ");
      printf("%d", tab[i]);
   }
   printf("]\n");
}

void test(int tab[], size_t taille) {
   printf("Avant inverser : \n");
   afficher(tab, taille);
   inverser(tab, tab + taille - 1);
   printf("Apres inverser : \n");
   afficher(tab, taille);
}
```

```
// Avant inverser :
// [1]
// Apres inverser :
// [1]
//
// Avant inverser :
// [1, 2]
// Apres inverser :
// [2, 1]
//
// Avant inverser :
// [1, 2, 3]
// Apres inverser :
// [3, 2, 1]
```

Solution exercice 2.14

1) 12, 6

2) 12, 24

3) 3, 6

4) 12, 24