



M2101 - Architecture 2

Cours n°1 :

Introduction au C

Victor Poupet

Premier programme

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Quelques généralités :

- sensible à la casse
- caractères blancs ne sont pas significatifs

Instructions

```
int a = 12;  
int b = 5;  
  
/* bloc d'instructions après if */  
if (a > 0) {  
    a += 1;  
    b = a + b;  
}
```

- Les instructions se terminent par un point virgule
- On peut grouper plusieurs instructions entre accolades

Compilation

- Les programmes en C ne sont pas directement exécutables
- Il faut les compiler
- La compilation produit un fichier exécutable (langage machine)

```
$ gcc -Wall code.c -o prog
```

```
$ ./prog  
Hello, world!
```

```
$
```

La compilation s'effectue en plusieurs temps :

- précompilation
(traitement syntaxique)
- la compilation
(génération de code machine)
- l'édition de liens
(production d'un exécutable en combinant les fonctions)

Précompilation

```
#include <stdio.h>
#include <stdlib.h>
#include "fichier.h"
```

```
#define PI 3.141592653589793
#define ever ;1;
```

```
int main() {
    int n = 1;
    for(ever) {
        if (PI/n < 1)
            break;
        n++;
    }
}
```

- Inclusion de fichiers (`#include`)
- Remplacement des constantes par leurs valeurs (`#define`)

Précompilation

```
#include <stdio.h>
#include <stdlib.h>
#include "fichier.h"
```

```
#define PI 3.141592653589793
#define ever ;1;
```

```
int main() {
    int n = 1;
    for(;1;) {
        if (3.141592653589793/n < 1)
            break;
        n++;
    }
}
```

- Inclusion de fichiers (`#include`)
- Remplacement des constantes par leurs valeurs (`#define`)

Variables

```
// variable globale
char *s = "youpi";

int f(int a) {
    int b; // variable locale
    b = 2*a;
    return a + b;
}
```

Les variables peuvent être

- globales : déclarées hors de toute fonction et partagées pour tout le programme
- locales : déclarées dans une fonction, valables uniquement dans cette fonction et ses sous-fonctions

Fonctions

```
#include <math.h>
```

```
float hyp(int, int);
```

```
float hyp(int a, int b) {  
    float r;  
    r = sqrt(a*a + b*b);  
    return r;  
}
```

Chaque fonction a un prototype :

- type de retour
- nom de la fonction
- liste des arguments : type éventuellement suivi du nom

Le prototype sert à définir le cadre d'utilisation de la fonction

Exemple : `stdio.h`

La fonction main

```
int main() {  
    // code principal du programme  
    return 0;  
}
```

```
int main(int argc, char *argv[]) {  
    // affiche tous les arguments :  
    for (int i=0; i<argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
}
```

Tout programme doit comporter une fonction `main` :

- c'est cette fonction qui est exécutée quand le programme est lancé
- la fonction `main` peut faire appel aux autres fonctions
- son type de retour est toujours un entier (code de retour du programme)
- elle peut recevoir des arguments correspondant aux options lors de l'appel :
 - un entier indiquant le nombre d'arguments
 - un tableau contenant des chaînes de caractères (les arguments)

Types de base

```
int a = 12;  
unsigned int b;  
long int c;  
long long int d;  
unsigned long long int e;
```

```
float x = -2.72;  
double y;
```

```
char s = 'A';
```

- Chaque variable doit être d'un type défini
- Le type permet au compilateur de déterminer l'espace mémoire occupé par la variable, et de décoder correctement sa valeur
- Il existe plusieurs types de données “élémentaires” prédéfinis

Les types de base en mémoire

Les différents types de base n'occupent pas le même espace mémoire.

En général :

- **char** sur un octet (minimum 1)
- **short** sur 2 octets (minimum 2)
- **int** sur 4 octets (minimum 2)
- **long int** sur 8 octets (minimum 4)
- **float** sur 4 octets
- **double** sur 8 octets

Cependant, l'espace occupé par chaque type peut dépendre du système d'exploitation. La norme spécifie des tailles minimum et des relations entre les types (un **long int** doit être au moins aussi long qu'un **int**)

La fonction **sizeof** renvoie la taille en octets d'un type (ex : **sizeof(long int)**)

Structures de contrôle

```
int x = 12;
```

```
if (x>0) {  
    x--;  
} else {  
    x++;  
}
```

```
while (x>1) {  
    x--;  
}
```

```
for (int i=0; i<5; i++) {  
    x *= 2;  
}
```

```
switch (x%3) {  
    case 1:  
        x -= 1;  
        break;  
    case 2:  
        x += 1;  
        break;  
    default:  
        x += 3;  
        break;  
}
```

On peut contrôler le déroulement du programme en fonction des valeurs des variables à l'aide de structures de contrôle :

- **if ... else** permet d'exécuter une instruction si une condition est vérifiée ou non
- **while** permet d'exécuter une instruction tant qu'une condition est vraie
- **for** est une syntaxe spéciale qui génère une boucle **while**
- **switch** permet d'exécuter des instructions en fonction de la valeur d'une expression

Un exemple

→ Programme de calcul de racine d'un polynôme de degré 2

Conversion de type

```
char c = 'A'; // 'A' = 65
int a = 3;
int b = -1;
unsigned int u;
float x, y;

x = a / b;
y = (float) a / b;
// x = 1, y = 1.5

a = (int) 3.9;
u = (unsigned short) b;
b = a + c;
// a = 3, u = 65535, b = 68
```

Dans certains cas, il est possible de convertir une variable d'un type en un autre (*cast*) :

- la plupart des types de base sont inter-convertibles car ils représentent tous des nombres
- pour demander la conversion explicite en un type, on préfixe la variable du type entre parenthèse
- certaines conversions sont faites implicitement lors d'opérations entre deux types différents
- la conversion d'un **float** en **int** élimine la partie fractionnaire (arrondi inférieur en positif, et supérieur en négatif)

Tableaux

```
/* déclarations */

int tab[10];
// le programme réserve 40 octets

long int tab2[10];
// le programme réserve 80 octets

char s[20];
// le programme réserve 20 octets

/* utilisation */

tab[0] = 0;
tab[1] = 1;
for (int i=2; i<10; i++) {
    tab[i] = tab[i-1] + tab[i-2];
}

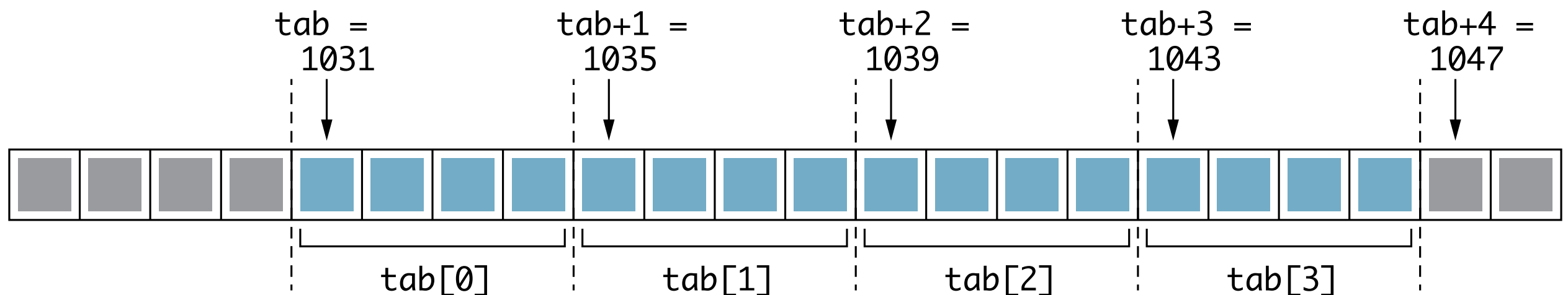
// initialisation à la création
int t[4] = {1, 2, 3, 4};
```

- Un tableau contient toutes les valeurs à la suite en mémoire ;
- L'espace mémoire d'un tableau est donc égal à la taille d'un élément multiplié par le nombre d'éléments ;
- La variable correspondant au tableau ne contient pas véritablement toutes les valeurs, mais simplement l'adresse mémoire où se trouve la première case du tableau ;
- Pour trouver la valeur en position *i* dans le tableau, il faut aller à l'adresse de la première case, et se déplacer de *i* fois la taille d'un élément.

Tableaux

Suite à l'instruction
`int tab[4];`

- Le programme réserve 16 octets (4 par entier) pour le tableau ;
- La variable **tab** contient l'adresse de la première case ;
- La valeur qui se trouve à l'adresse **tab** est un **int** ;
- La valeur qui se trouve à l'adresse **(tab+1)** est un **int** ;
- Rien ne marque la fin du tableau, mais le contenu des cases à l'extérieur est indéterminé (et peut être interdit d'accès au programme courant).



Pointeurs

```
int a, b, *p;
```

```
a = 10;
```

```
p = &a;
```

```
// *p vaut 10
```

```
b = *p;
```

```
a++;
```

```
// a = 11, *p = 11, b = 10
```

- Un pointeur est une variable qui contient l'adresse mémoire d'un objet (et qui connaît le type de l'objet pointé)
- La notation `*p` désigne la valeur qui se trouve à l'adresse `p`
- La notation `&a` désigne l'adresse à laquelle est stocké l'objet `a`
- Dès que l'on manipule des objets plus complexes que les types de base, les variables contiennent des pointeurs vers les objets et non pas les objets eux-mêmes

Pointeurs

```
float a = 3.14;
```

```
(float) a : 1234
```



Pointeurs

```
float a = 3.14;  
float *p;
```

```
(float) a : 1234  
(float*) p : 1842
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;
```

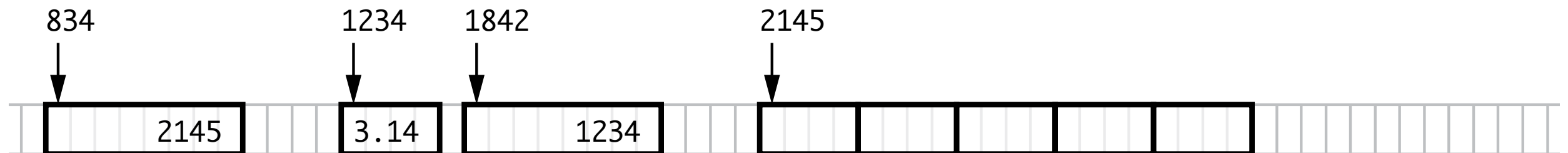
```
(float) a : 1234  
(float*) p : 1842
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];
```

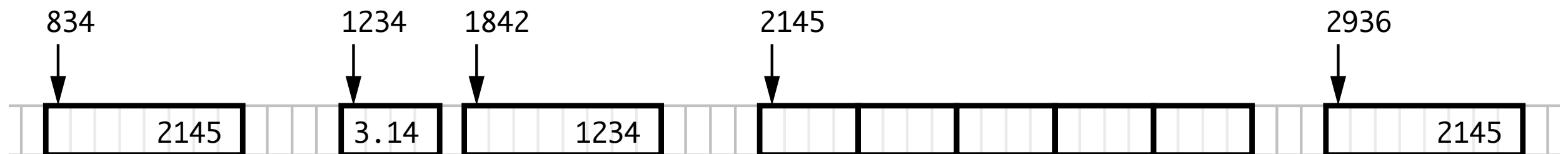
```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];  
int *t = tab;
```

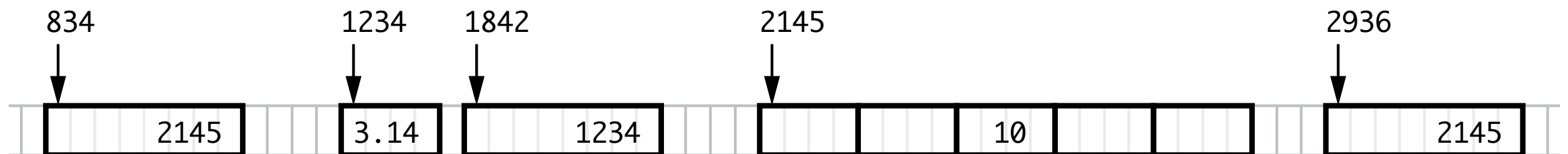
```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834  
(int*) t : 2936
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];  
int *t = tab;  
tab[2] = 10;
```

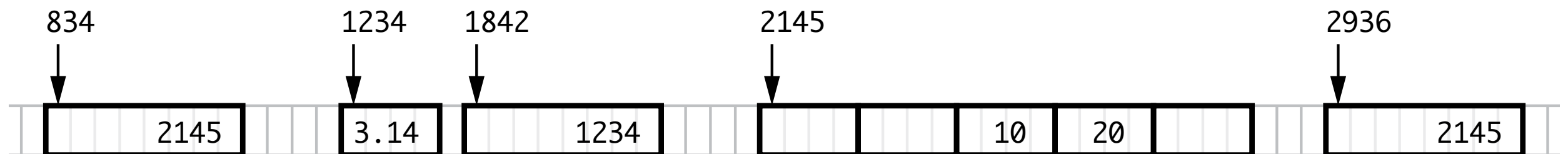
```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834  
(int*) t : 2936
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];  
int *t = tab;  
tab[2] = 10;  
*(t+3) = 20;
```

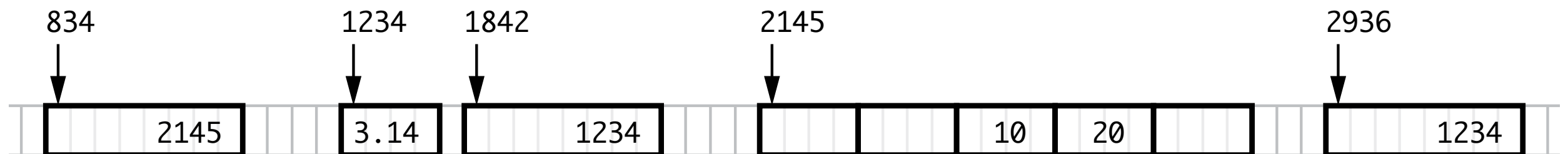
```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834  
(int*) t : 2936
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];  
int *t = tab;  
tab[2] = 10;  
*(t+3) = 20;  
t = p;
```

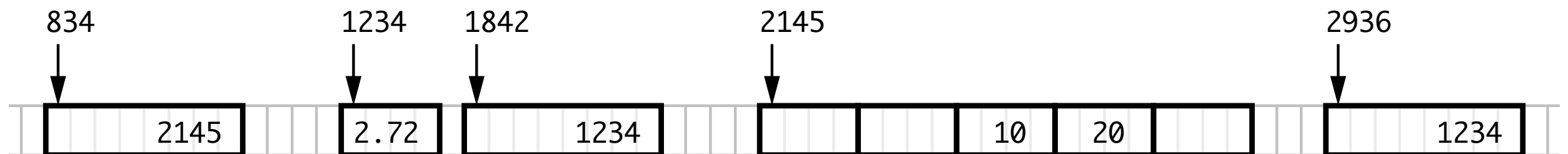
```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834  
(int*) t : 2936
```



Pointeurs

```
float a = 3.14;  
float *p;  
p = &a;  
int tab[5];  
int *t = tab;  
tab[2] = 10;  
*(t+3) = 20;  
t = p;  
*p = 2.72;
```

```
(float) a : 1234  
(float*) p : 1842  
(int*) tab : 834  
(int*) t : 2936
```



Valeur ou référence

```
int a, b;  
a = 1;  
b = a;
```

```
a = 2;  
// a = 2, b = 1
```

```
int u[3] = {0, 10, 20};  
int *v;  
v = u;
```

```
u[1]++;  
// u[1] = 11, v[1] = 11
```

```
v++;  
// u[1] = 11, v[1] = 20
```

```
char *n, *m;  
n = "Toto";  
m = n;
```

```
n = "Titi";  
// n = "Titi", m = "Toto"
```

- Lorsqu'on copie une variable contenant une valeur de type simple (**int**, **char**, **float**, etc.), la valeur est copiée et les deux variables sont indépendantes
- Si deux pointeurs sont égaux, toute modification du contenu de l'un modifie également le contenu de l'autre

Pointeurs comme arguments

```
void f(int n) {  
    n++;  
}  
  
void g(int *p) {  
    (*p)++;  
}  
  
int main() {  
    int n = 10;  
  
    f(n);  
    // n = 10  
  
    g(&n);  
    // n = 11  
}
```

- Si l'on passe un entier (ou un autre type simple) à une fonction, c'est une copie de la valeur qui est transmise
- Si on doit modifier la valeur de cet entier, il faut passer un pointeur vers la valeur
- Plusieurs fonctions en C prennent un argument qui est un pointeur vers un emplacement où écrire le résultat de leur exécution (ex : `scanf`)

Tableaux et pointeurs

Les opérations sur les tableaux reviennent à manipuler des pointeurs :

- la déclaration `int t[];` est équivalente à `int *t;` (dans ce cas la taille du tableau n'est pas définie et l'espace n'est pas réservé en mémoire)
- `t[5]` est équivalent à `*(t+5)` (sauf lors d'une déclaration)

Les chaînes de caractères

```
int longueur (char *s) {  
    int i = 0;  
    while (s[i] != '\0') {  
        i++;  
    }  
    return i;  
}
```

```
int main() {  
    char *s = "Youpi";  
    int l = longueur(s);  
    // l = 5  
  
    char nom[5];  
    nom[0] = 'T';  
    nom[1] = 'o';  
    nom[2] = 't';  
    nom[3] = 'o';  
    nom[4] = '\0';  
    l = longueur(nom);  
    // l = 4  
}
```

- Les chaînes de caractères sont des tableaux de caractères, dont la dernière case contient '\0'
- En général il n'est donc pas nécessaire de mémoriser la longueur d'une chaîne de caractères
- Lorsque l'on réserve l'espace mémoire manuellement, il faut faire attention à prendre une case de plus pour le caractère '\0'

Structures

```
// définition du type
struct Voiture {
    int nb_portes;
    char *marque;
    float consommation;
};
```

```
// initialisation
struct Voiture caisse;
caisse.marque = "Peugeot";
caisse.nb_portes = 5;
caisse.consommation = 6.3;
```

```
struct Voiture bagnole = {
    .nb_portes = 3,
    .consommation = 7.1,
    .marque = "Volvo",
};
```

```
struct Voiture chariot =
    {2, "Fiat", 5.9};
```

Les structures sont des objets en C contenant plusieurs *champs*.

- La définition d'une structure donne un type et un nom à chaque champ
- On accède aux champs à l'aide de la notation pointée
- Les valeurs des champs d'une structure sont stockés à la suite en mémoire
- La taille occupée en mémoire par une structure est égale à la somme des tailles des différents champs
- On peut utiliser `sizeof` pour trouver la taille