LE LANGAGE C

REVISION DES POINTEURS ET LIGNE DE COMMANDE

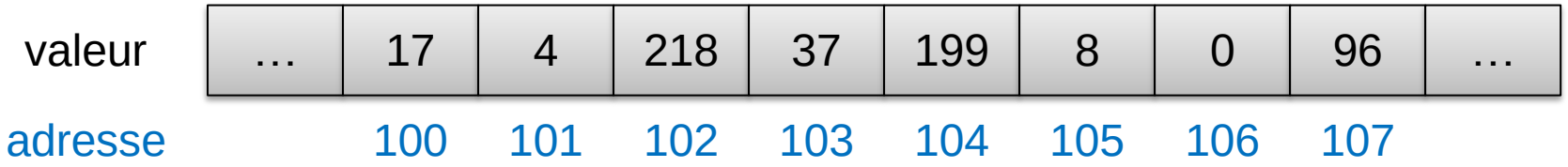
F. Glück, V. Pilloux

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

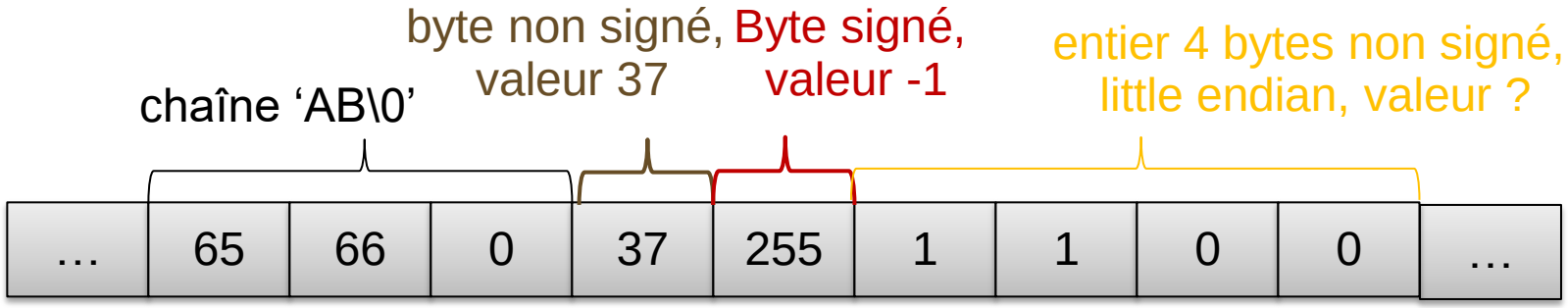
Mémoire

- Pour un programme C, la mémoire est simplement une suite d'octets ou de *bytes*
- Chaque byte a une valeur et une **adresse** en mémoire



Représentation de la mémoire

- Ces bytes peuvent avoir des représentations différentes. Par exemple:
  - 1 byte: nombre entier de 0 à 255 en représentation non signée ou de -128 à +127 en représentation signée
  - 1 byte: code ASCII d'un caractère (le code ASCII de 'A' est 65)
  - 2 bytes: mot représentant un nombre entier entre 0 et 65535 (si non signé) ou -32768 et +32767 (si signé)
  - 4 bytes … etc…
- Pour les nombres à plusieurs bytes, l'ordre des bytes en mémoire dépend de l'architecture du processeur:
  - Little endian: byte de poids faible en premier (cas de Intel, ARM)
  - Big endian: byte de poids fort en premier



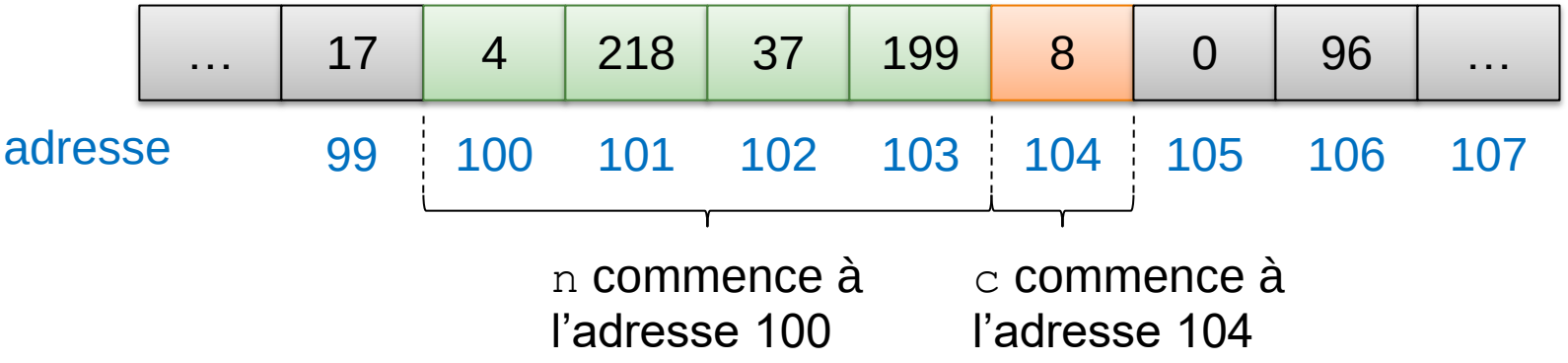
Mémoire et variables

- Quand on définit des variables:

```
int n;
unsigned char c;
```

- Deux chosent se produisent…

  1. De la mémoire est réservée pour stocker les variables
  2. Le compilateur stocke leurs **adresses**



Mémoire et variables

- Quand on définit des variables:

```
int n;
unsigned char c;
```

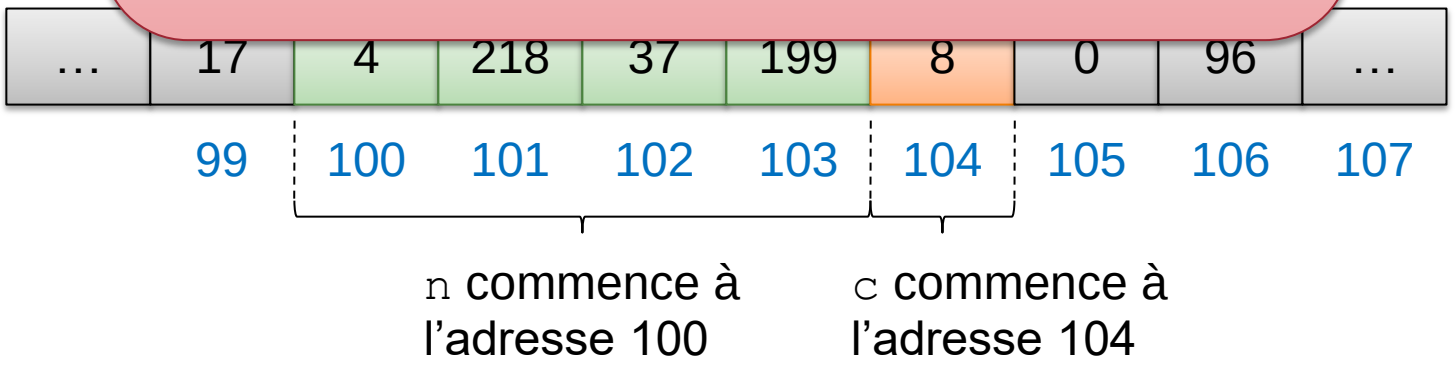
- De

1.
2.

ariables

Une variable est donc caractérisée par:

- Sa valeur
- Son adresse en mémoire



Pointeurs (1)

- Un pointeur est une **variable** contenant une **adresse** comme valeur
- L'astérisque ***** est utilisée lors de la déclaration d'une variable pour indiquer que celle-ci est un pointeur

```
int *ptr;                  // pointeur sur un entier

unsigned char *ch;         // pointeur sur un caractère
                           // non signé

char *str;                 // pointeur sur une chaîne
                           // de caractères

int **p;                   // pointeur sur le pointeur
                           // d'un entier (double pointeur)
```

Pointeurs (2)

- L'adresse d'une variable est obtenue avec l'opérateur **&**
- L'adresse renvoyée forme un pointeur sur le type de la variable

```
int i = 8;          // Entier

int *p = &i;        // p pointe sur i

double *d = &i; // Erreur: pointeur sur un double !
```

Pointeurs (3)

- Un pointeur est déréférencé avec l'opérateur *****
- ***p** renvoie la **valeur** se trouvant à l'adresse pointée par le pointeur p
- Utilisé pour lire ou changer la valeur pointée

* Indique une déclaration de pointeur

* Indique un déréférencement

```
int i = 8;      // Entier
int *p = &i;    // p pointe sur i
int j = *p;     // j vaut maintenant 8
*p = 12;        // i vaut maintenant 12
```

- **Pour pouvoir être utilisé (déréférencé), un pointeur doit toujours pointer sur un espace mémoire déjà alloué!**

Pointeurs (4)

- On peut donc écrire :

```
int x;

int *p = &x;
```

- Le contenu de `x` peut alors être affecté de deux manières différentes :

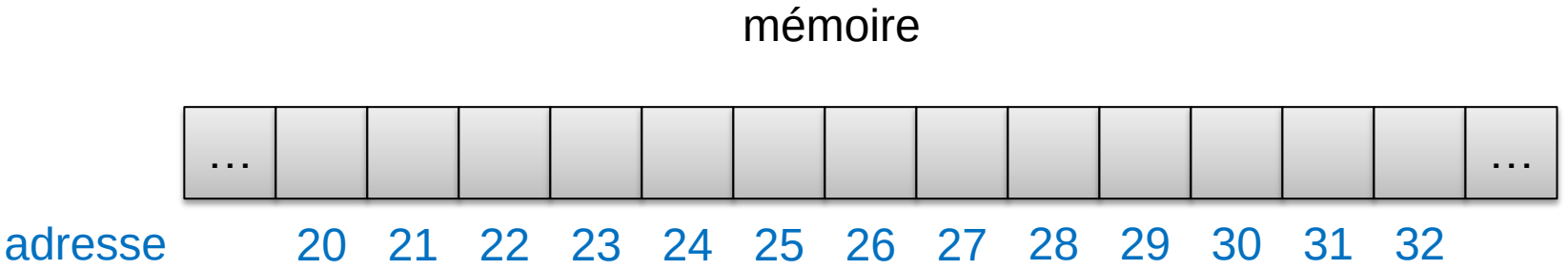
```
x = 7;

*p = 7;
```

  Où `*p` signifie la valeur pointée par `p`, qui n'est rien d'autre que la valeur de `x`.

- A noter que utilisés en cascade (à éviter), les opérateurs * et & s'annulent:

```
*&a = a
```

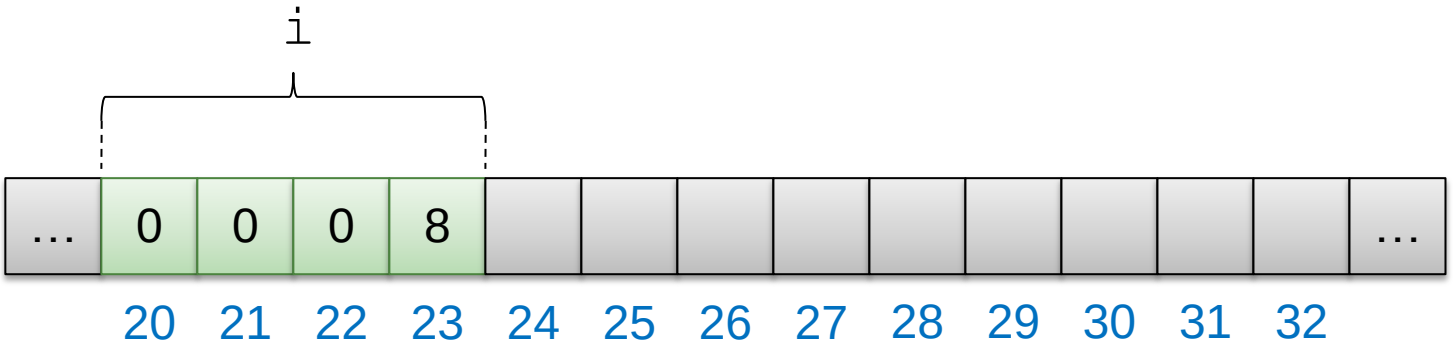
Pointeurs: exemple




```
int i = 8;
```

En admettant que le compilateur alloue de la mémoire à partir de l'adresse 20 pour cette variable et qu'il la stocke en format big-endian, quel sera le contenu de la mémoire après l'exécution de la ligne?

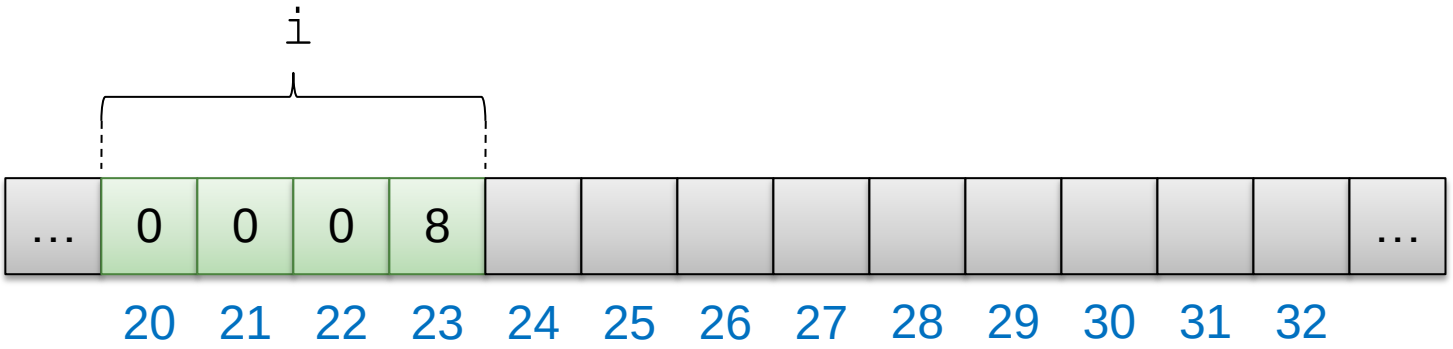
Pointeurs: exemple




```
int i = 8;
```

&i égal à 20

Pointeurs: exemple



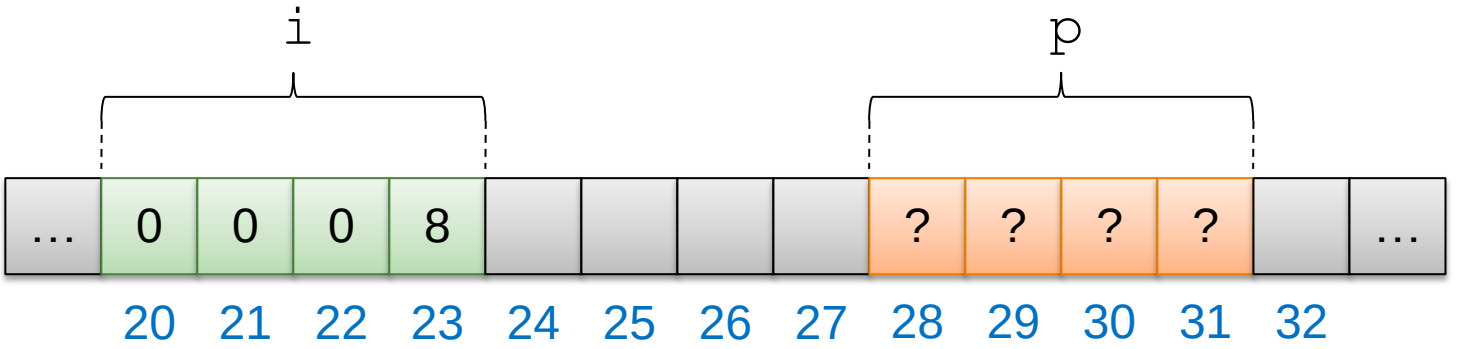
Même question pour p
(avec allocation en 28)?

```
int i = 8;
```

&i égal à 20

```
int *p;
```

Pointeurs: exemple

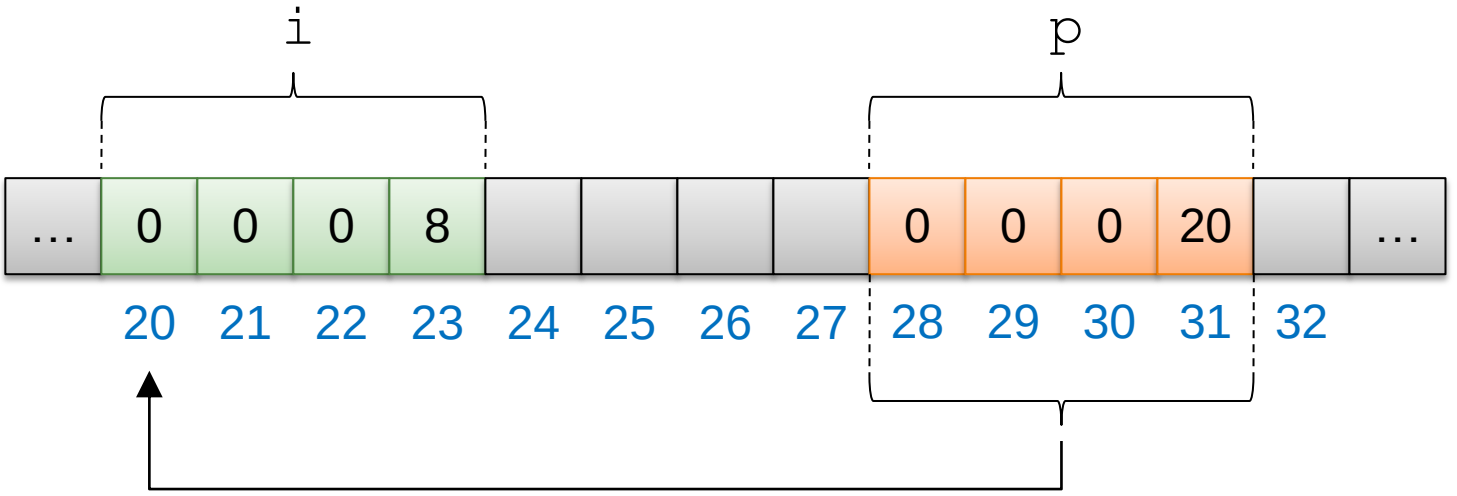



```
int i = 8;
```
&i égal à 20

```
int *p;
```
&p égal à 28

Adresses imposées par le compilateur ici

Pointeurs: exemple




```
int i = 8
```

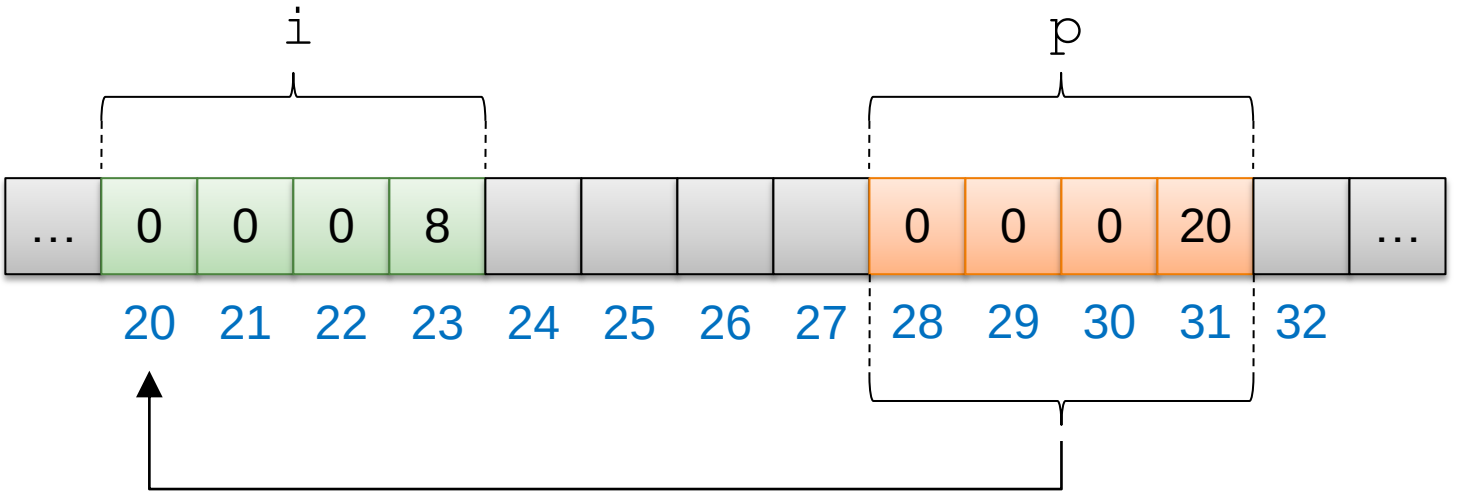
&i égal à 20

```
int *p = &i (= 20)
```

&p égal à 28

Attention: il ne s'agit pas d'un déréférencement, mais de la déclaration d'un pointeur!

Pointeurs: exemple




```
int i = 8;
```

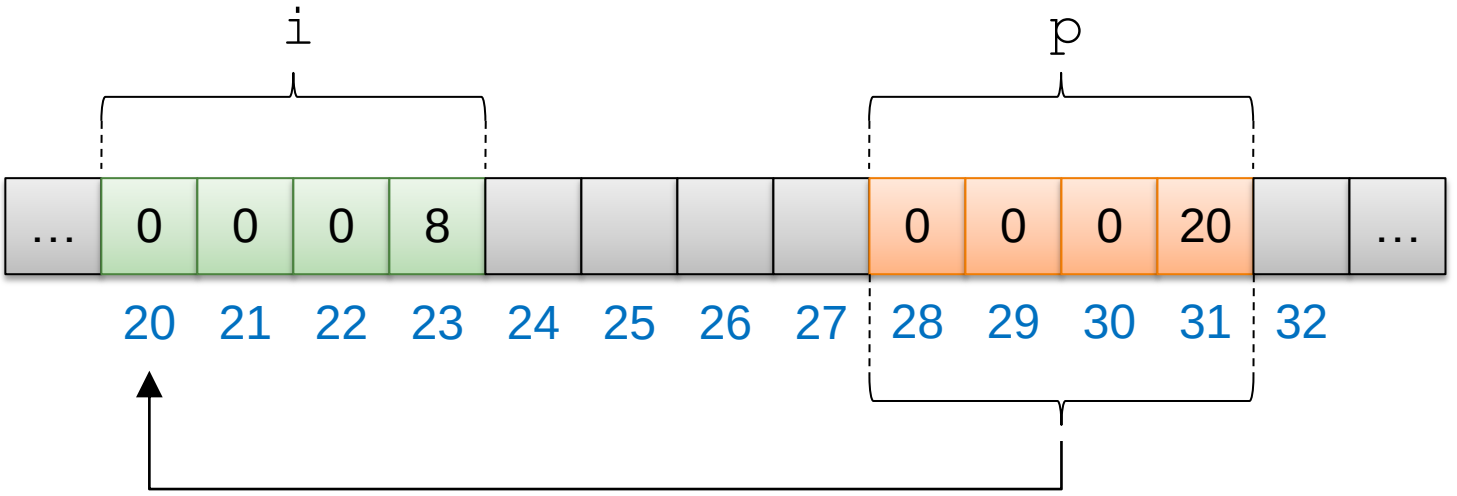
&i égal à 20

```
int *p = &i; (= 20)
```

&p égal à 28

```
int j = *p = ?;
```

Pointeurs: exemple




```
int i = 8;
```

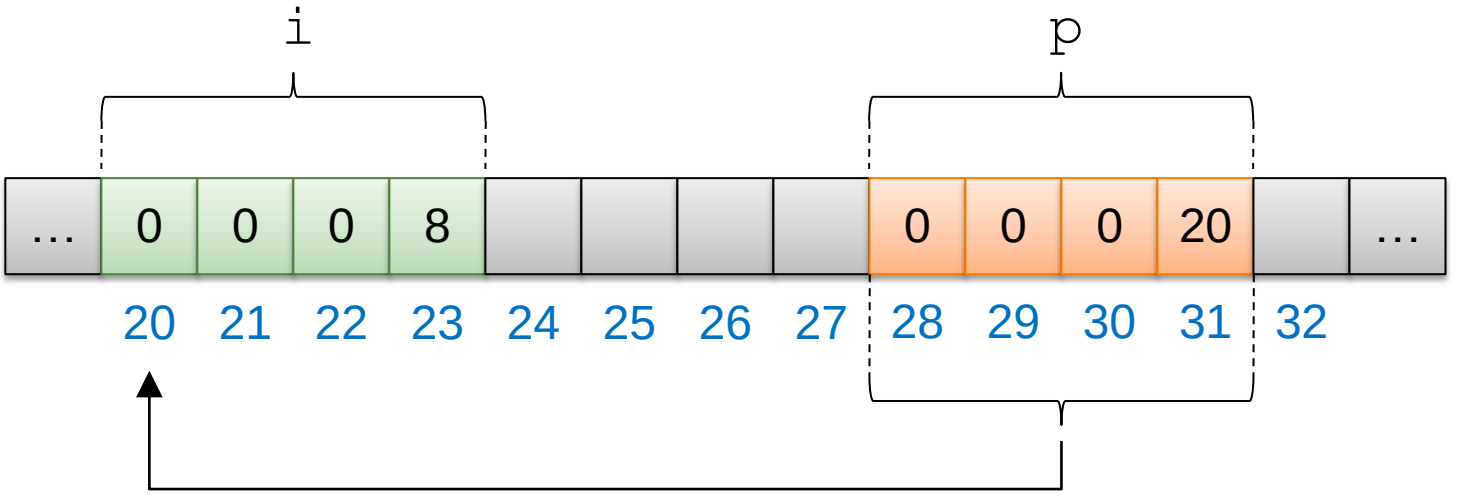
&i égal à 20

```
int *p = &i; (= 20)
```

&p égal à 28

```
int j = *p; (= 8)
```

Pointeurs: exemple




```
int i = 8;
```

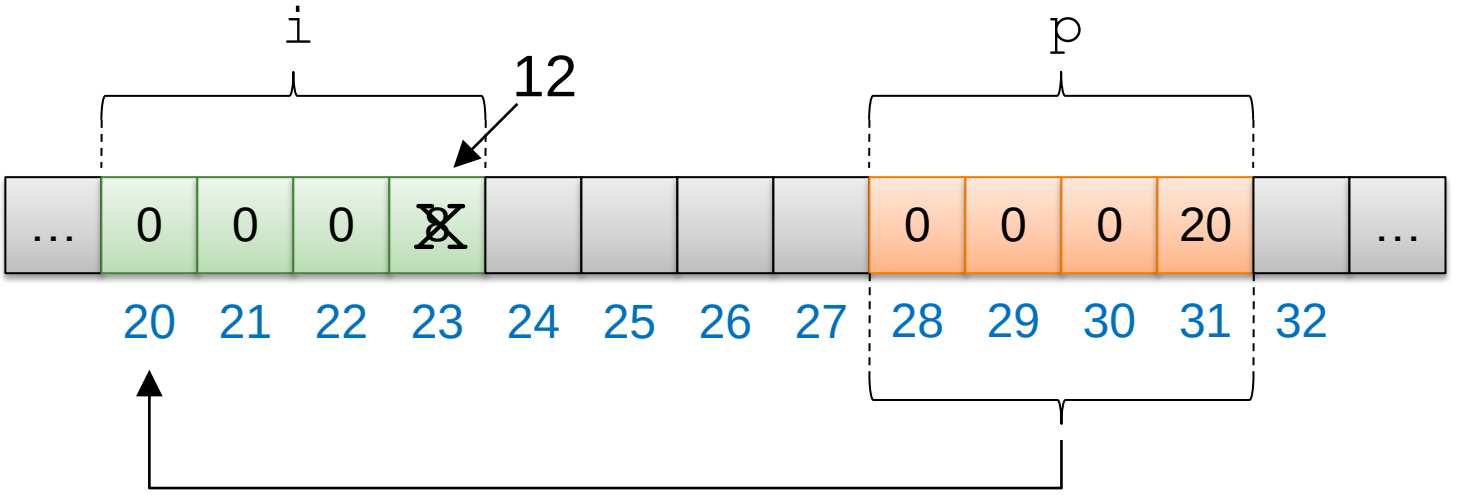
&i égal à 20

```
int *p = &i; (= 20)
```

&i égal à 28

```
int j = *p; (= 8)
*p = 12;
```

Pointeurs: exemple




```
int i = 8;
```

&i égal à 20

```
int *p = &i; (= 20)
```

&p égal à 28

```
int j = *p; (= 8)
*p = 12;
```

Pointeurs et arguments (1)

- C supporte **uniquement** le passage de paramètre par **valeur !**
- **Ces valeurs sont jetées dès la sortie de la fonction**
- Passer l'adresse d'une structure en argument permet de modifier les valeurs de la structure passée par l'appelant

```
// Lent (copie complète de la structure)
// Structure modifiable dans la fonction
// mais perte des modifications en sortant
void slow(struct LargeStructure p) {
    p.x = 10;
    p.y = 20;
}

// Rapide, copie de l'adresse uniquement
// Structure modifiable dans la fonction mais
// plus de perte des modifications en sortant
void fast(struct LargeStructure *p) {
    p->x = 10;
    p->y = 20;
}
```

Pointeurs et arguments (2)

- Les pointeurs sont nécessaires lorsqu'on désire **modifier la valeur des arguments**:

```
// Permute les valeurs de a et b
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    int x = 10, y = 7;
    swap(&x, &y);
}
```

- **Sans le passage des adresses en argument, la fonction `swap` n'aurait aucun moyen de modifier les valeurs des arguments!**

Pointeurs et arguments (3)

L'utilité des pointeurs dans le cas d'arguments est aussi de permettre à des fonctions d'accéder aux données elles mêmes et **non à des copies**.

Dans l'exemple précédent de la fonction permutant la valeur de deux entiers :

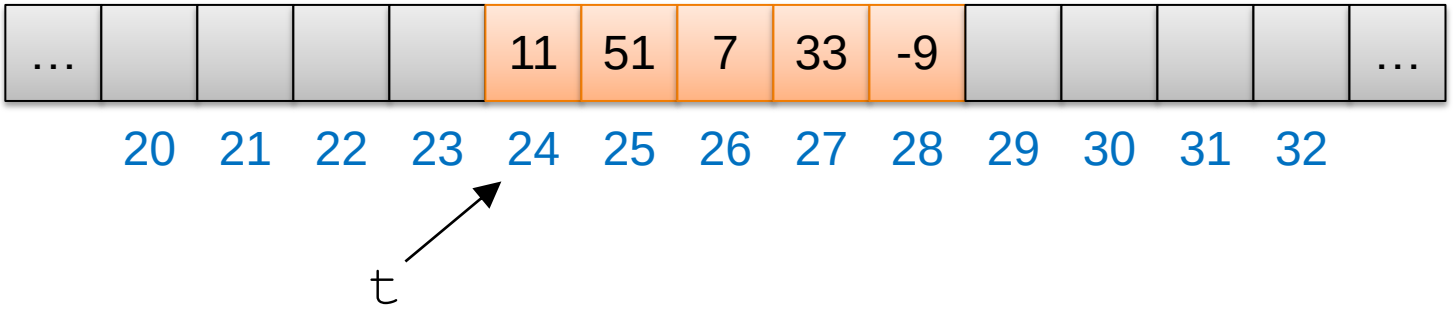
```
// Permute les valeurs de a et b
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main(void) {
    int x = 10, y = 7;
    swap(&x, &y);
}
```

On passe donc l'adresse des variables `x` et `y` comme paramètres puisque la fonction attend des pointeurs comme paramètres. Il s'agit donc bien d'un passage par valeur. D'ailleurs les valeurs des pointeurs passés sont jetées au retour de la fonction.

Pointeurs et tableaux (1)

```
char t[5] = { 11, 51, 7, 33, -9}, *p=t;
```



- `t` n'est rien d'autre que l'adresse du premier élément du tableau !
- Ecrire `t` est équivalent à écrire `&t[0]`
- **Donc t n'est rien d'autre qu'un pointeur sur un tableau!**
- Seule différence entre t et p, c'est que t a une valeur **constante** alors que p peut être modifié:

p = t+10; // OK t = t+10; // interdit

Pointeurs et tableaux (2)

- Un tableau peut-être passé de deux manières différentes en argument :

```
void func1(char t[], int size) {
    …
}

void func2(char *t, int size) {
    …
}

void main(void) {
    char table[3] = { 1, 2, 3};
    func1(table, 3);
    func2(table, 3);
}
```

- Y-a-t-il une différence entre ces 2 prototypes ?

Pointeurs et tableaux (2)

- Un tableau peut-être passé de deux manières différentes en argument :

```
void func1(char t[], int size) {
    …
}

void func2(char *t, int size) {
    …
}

void main(void) {
    char table[3] = { 1, 2, 3};
    func1(table, 3);
    func2(table, 3);
}
```

- Y-a-t-il une différence entre ces 2 prototypes ?
- **Non**, le compilateur remplace l'expression `t[]` par `*t`

Arithmétique des pointeurs (1)

- Le C gère automatiquement l'arithmétique des pointeurs
- La sémantique **dépend de la taille du type pointé !**
- L'opérateur `sizeof` permet de connaître la taille d'un type en bytes:

`sizeof(int)` ⇨ taille d'un entier

`sizeof(unsigned char)` ⇨ taille d'un caractère non signé

`sizeof(struct Complex)` ⇨ taille de la structure Complex

`sizeof(int*)` ⇨ taille d'un pointeur sur un entier (dépend du système)

Arithmétique des pointeurs (2)

- Incrémenter un pointeur signifie pointer sur l'élément suivant en mémoire ⇨ dépend donc de la taille du type pointé !

```
int *p = &x;
p = p+1;   // incrémente p pour pointer à l'élément
           // suivant (avance de sizeof(int) bytes
           // en mémoire)
struct Complex *cp = &c;
cp++;      // incrémente cp pour pointer sur l'élément
           // suivant (++ -> +sizeof(struct Complex))
```

Arithmétique des pointeurs (3)

- Cela devient évident lorsqu'un pointeur est utilisé comme tableau :

```
int i;
int tab[5] = { 1,2,3,4,5 };
int *p = tab;

*p = 7;        // quel élément du tableau est affecté?

for (i = 0; i < 5; i++) {
    printf("%d\n", *p);
    p++;   // positionne le pointeur sur l'élément
}          // suivant du tableau. Quel est l'incrément?
```

Arithmétique des pointeurs (4)


```
int tab[5];
```

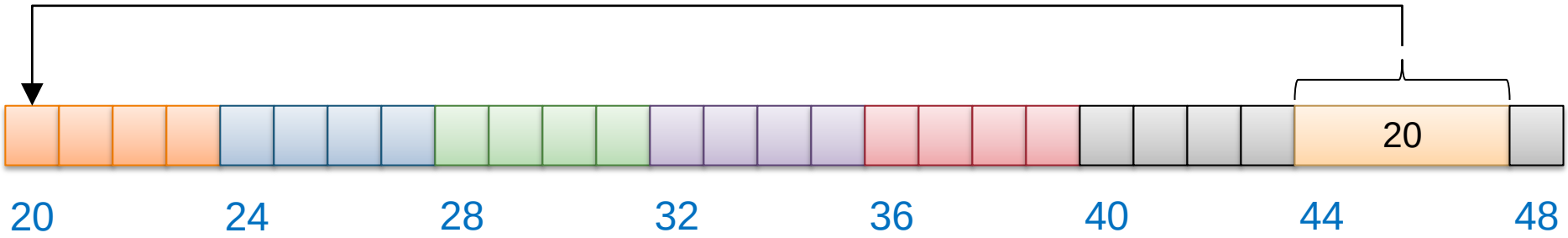
Arithmétique des pointeurs (4)


```
int tab[5];
```

Arithmétique des pointeurs (4)

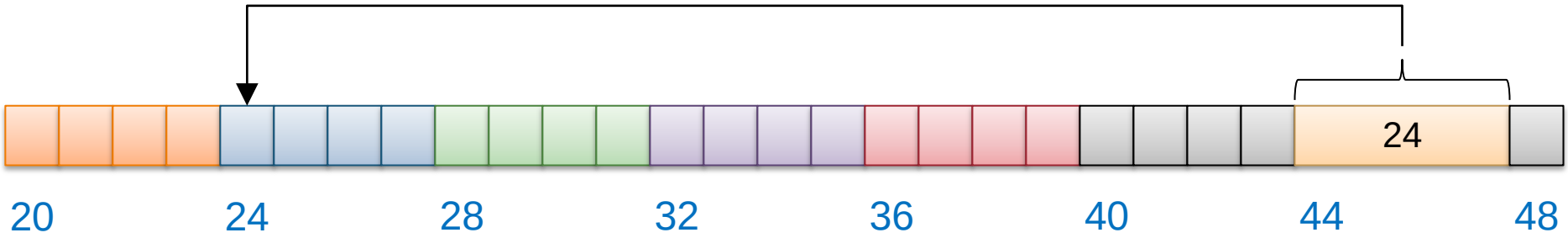

```
int tab[5];
int *p;
```

Arithmétique des pointeurs (4)




```
int tab[5];
int *p = tab;
```

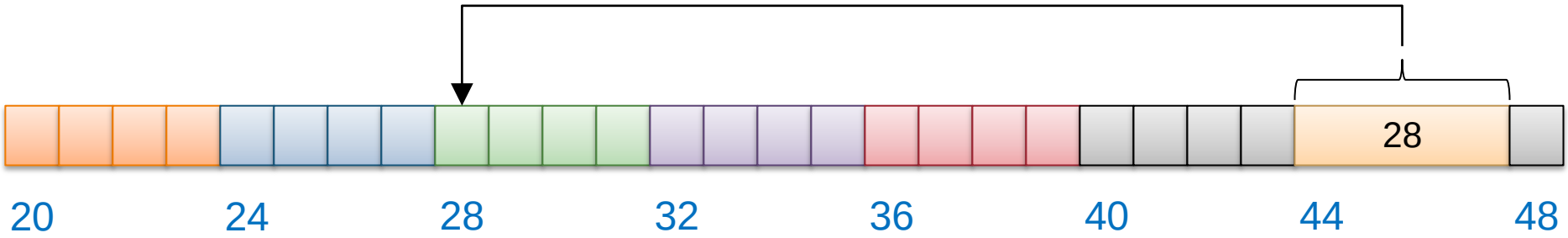
Arithmétique des pointeurs (4)




```
int tab[5];
int *p = tab;
```

```
p++;
```

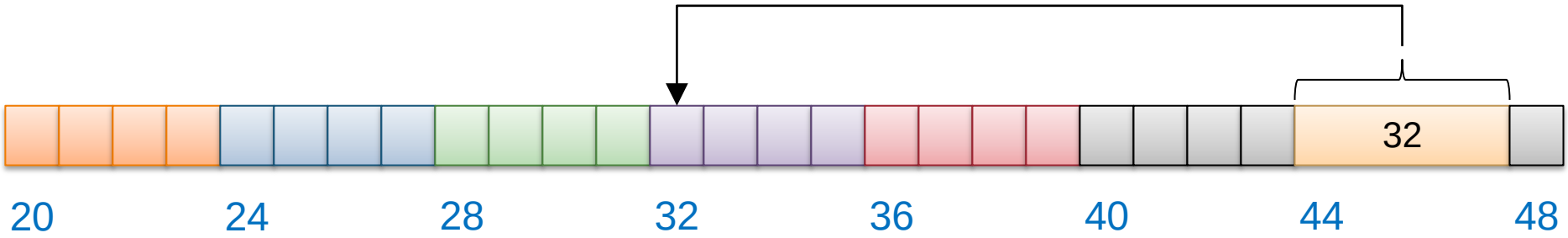
Arithmétique des pointeurs (4)




```
int tab[5];
int *p = tab;
```

```
p++;
p++;
```

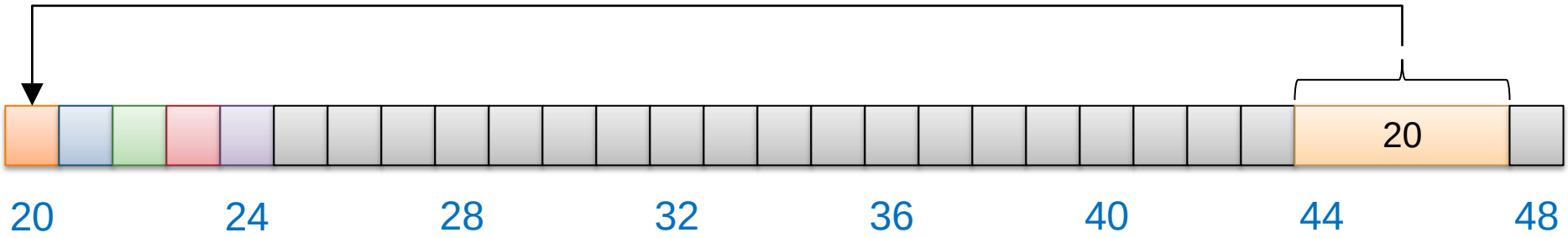
Arithmétique des pointeurs (4)




```
int tab[5];
int *p = tab;
```

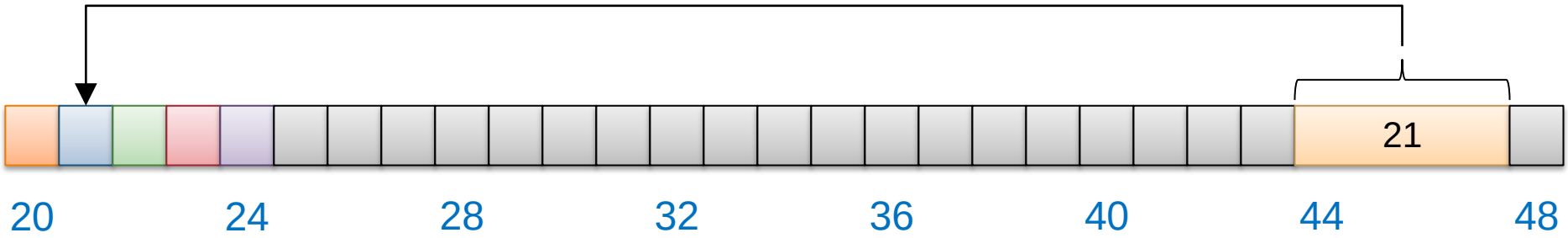
```
p++;
p++;
p++;
```

Arithmétique des pointeurs (5)




```
char tab[5];
char *p = tab;
```

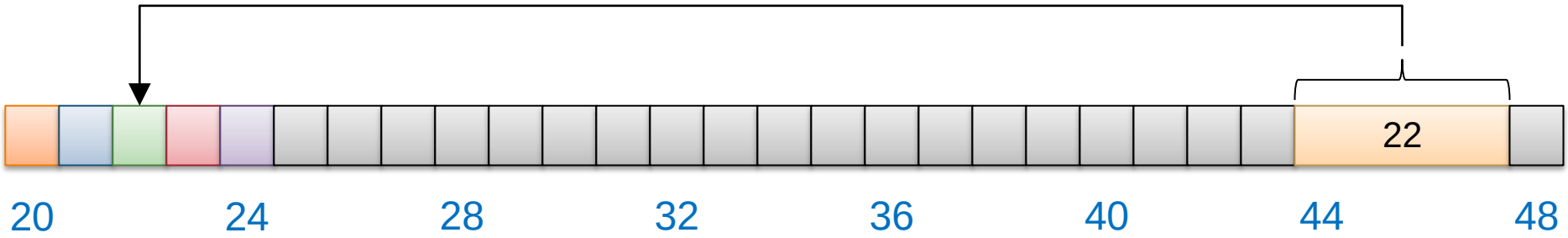
Arithmétique des pointeurs (5)




```
char tab[5];
char *p = tab;
```

```
p++;
```

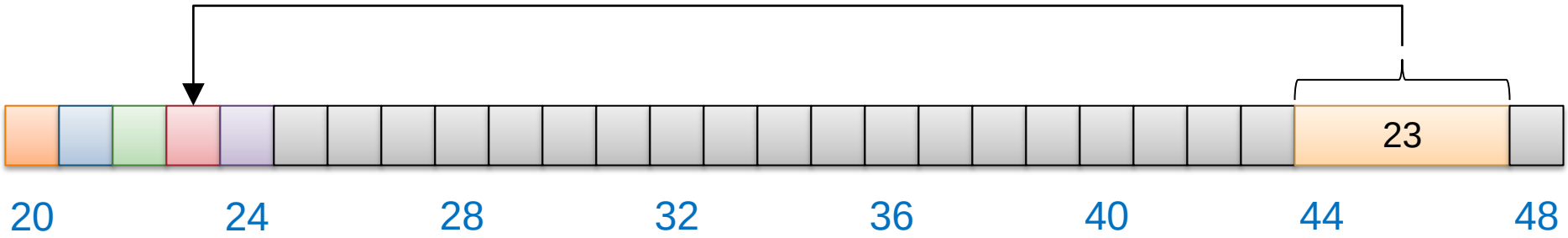
Arithmétique des pointeurs (5)




```
char tab[5];
char *p = tab;
```

```
p++;
p++;
```

Arithmétique des pointeurs (5)




```
char tab[5];
char *p = tab;
```

```
p++;
p++;
p++;
```

Pointeurs et tableaux (1/2)

- Un tableau peut-être vu comme un pointeur
- Indexer un tableau revient simplement à faire de l'arithmétique de pointeurs:

```
tab[i]
```

est donc équivalent à:

```
*(tab + i)
```

- De fait, la formule calculant l'adresse de `ptr+i` où `ptr` est de type `T*` devient alors:

```
addr(ptr + i) = addr(ptr) + [sizeof(T) * i]
```

Pointeurs et tableaux (2/2)

- Les trois codes suivants sont donc équivalents:

```
int i;
int tab[5] = { 1,2,3,4,5 };
int *p = tab;

for (i = 0; i < 5; i++) {
    printf("%d\n", p[i]);
}
```

```
int i;
int tab[5] = { 1,2,3,4,5 };
int *p = tab;

for (i = 0; i < 5; i++) {
    printf("%d\n", *(p+i));
}
```

```
int i;
int tab[5] = { 1,2,3,4,5 };
int *p = tab;

for (i = 0; i < 5; i++) {
    printf("%d\n", tab[i]);
}
```

Pointeurs: pré/post incrémentations (1)

- On peut incrémenter un pointeur de 3 manières différentes:
  - `p = p + 1`
  - `p++`
  - `++p`

- Attention aux instructions de pré/post incrémentation référençant la valeur pointée par un pointeur!

L'instruction:

```
if (*x++) {...}
```

est décomposable en:

```
int val = *x;
x = x+1;
if (val) {...}
```

L'instruction:

```
if (*++x) {...}
```

est équivalente à:

```
x = x+1;
int val = *x;
if (val) {...}
```

Pointeurs: pré/post incrémentations (2)

- Exemple de code copiant la zone mémoire pointée par `dst` à l'adresse de `src`:

```
void copy_array(int *src, int *dst, int size) {

   // Boucle sur la longueur (de size à zéro)
   while (size-- > 0) {
      // Copie l'élément pointé par src dand dst
      // puis incrémente les 2 pointeurs.
      // Equivalent à écrire:
      //    *dst = *src;
      //    dst++;
      //    src++;
      *dst++ = *src++;
}
```

Pointeurs: pré/post incrémentations (3)

- La syntaxe suivante incrémente le pointeur ou la valeur pointée selon les parenthèses:

```
int x[3]={12, 56, -3}, y1, y2, y3;
int *p=x;
y1 = *p++;    // incrément du pointeur
y2 = (*p)++; // incrémente la valeur pointée
y3 = ++(*p); // incrémente la valeur pointée
```

Quelle sont les valeurs y1, y2 et y3 à la fin du programme?

y1 = 12
y2 = 56
y3 = 58

Quiz (1)

- Que va afficher le programme suivant ?

```
void main() {
    int i = 5, j = 3;
    int *p, *q;
    p = &i;
    q = &j;
    j = 6;
    printf("%d %d", *p, *q);
}                                    // 5 6
```

- Que va afficher le programme suivant ?

```
void main() {
    int t[3] = {5, -9, 85};
    int *p=t;
    printf("%d\n", *++p);            // -9
    printf("%d\n", *p++);            // -9
    printf("%d\n", --(*p));          // 84
}
```

Quiz (2)

- Que va afficher le programme suivant ?

```
void func(char *str, short int *tab) {
        printf("%d\n", sizeof(str[0]));   // 1
        printf("%c\n", str[4]);           // s
        printf("%d\n", sizeof(str));      // 4  (ARM Cortex 3!)
        printf("%d\n", sizeof(tab[0]));   // 2
        printf("%d\n", sizeof(tab));      // 4  (ARM Cortex 3!)
}

void main() {
        char str[] = "pas simple les strings!";
        short int tab[] = {1,2,3};
        func(str, tab);
}
```

Allocation mémoire dynamique

- L'allocation dynamique de la mémoire se fait avec la fonction `malloc`:

```
#include <stdlib.h>

void *malloc(size_t size);
```

- Renvoie un pointeur sur la zone mémoire allouée ou NULL en cas d'échec
- La taille de la zone mémoire à allouer est **spécifiée en [byte]**!
- Afin d'être générique, la fonction renvoie un pointeur de type **`void`**, il est donc nécessaire de le « *caster* » au type désiré
- L'allocation dynamique est utile lorsqu'on **ne connait pas à l'avance** la taille de la mémoire à allouer (rappel: la déclaration d'un tableau est toujours de taille fixe!)

Libération mémoire

- La libération de la mémoire se fait avec la fonction `free`:

```
#include <stdlib.h>
void free(void *ptr);
```

- Ne pas oublier de libérer la mémoire!
- A chaque malloc() doit correspondre un appel de free()

Exemple d'allocation dynamique

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

int main(int argc, char *argv[]) {
        int table_size = 10;
        int *table = (int *) malloc(sizeof(int) * table_size);
        if (table == NULL) {
                fprintf(stderr, "malloc() failed!\n");
                return EXIT_FAILURE;
        }

        int i;
        for (i = 0; i < table_size; i++) {
                table[i] = i;
                printf("%d\n", table[i]);
        }
        free(table);
        table = NULL;
        return EXIT_SUCCESS;
}
```

Ligne de commande et pointeurs avancés

Florent Gluck

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Ligne de commande et arguments

- Point d'entrée d'un programme en C :

```
int main(int argc, char **argv)
```

- `argc` contient le nombre d'arguments passés sur la ligne de commande
- `argv` est un tableau de pointeurs (= un tableau de strings) contenant les arguments passés au programme sur la ligne de commande
- Le premier argument, `argv[0]`, est toujours le nom du programme
- Si on exécute le programme `xyx` avec les arguments `-n 2 file.txt`, alors `argc` contient la valeur 4 et `argv` contient:

```
argv[0] → "@xyz"
argv[1] → "-n"
argv[2] → "2"
argv[3] → "file.txt"
```

- **ATTENTION** : chaque argument est toujours une chaîne de caractères !

Conversion nombres ↔ strings

- Les arguments d'un programme sont toujours stockés sous forme de chaînes de caractères.
- Pas pratique si on veut manipuler des nombres.
- Solution: convertir les arguments (chaînes de caractères) en nombres.
- Fonctions de conversions:
  - `int atoi(const char *nptr);`
  - `long atol(const char *nptr);`
  - `long long atoll(const char *nptr);`
  - `double atof(const char *nptr);`
  - `sprintf(buf, "%d", 1234);  // où buf est un tableau de caractères préalablement alloué de taille suffisante`

Pointeurs de fonctions (1)

- Soit la fonction `max` retournant la valeur maximum d'un tableau d'entiers :

```
int max(int *t, int n) {
    int i = 1;
    int tmax = t[0];
    for(; i < n; i++) {
        if(t[i] > tmax)
            tmax = t[i];
    }
    return tmax;
}
```

- La valeur renvoyée par le nom d'une fonction étant l'adresse de son code en mémoire, nous pouvons l'affecter à un pointeur.

Pointeurs de fonctions (2)

- Déclaration du pointeur `pmax` devant pointer sur une fonction possédant le prototype (int*, int) :

```
int(*pmax)(int*, int);
```

- Le pointeur doit être déclaré avec la signature de la fonction, c'est-à-dire avec les types de paramètres corrects.
- A ce pointeur on affecte l'adresse d'une fonction ayant la même signature et celui-ci s'utilise alors comme s'il s'agissait d'une fonction:

```
pmax = max;
int tab[] = {11,86,1329,-13,457};
printf("%d", pmax(tab, 5));
```

Pointeurs et const

Soit : `int n = 77;`

Quelle sont les différences entre ?

```
a) const int *p = &n;
b) int const *p = &n;
c) int *const p = &n;
d) const int *const p = &n;
```

Pointeurs et const

Soit : `int n = 77;`

Quelle sont les différences entre ?

```
a) const int *p = &n;
b) int const *p = &n;
c) int *const p = &n;
d) const int *const p = &n;
```

a. la valeur pointée par `p`, `*p` est constante et ne peut être modifiée; donc `*p = 10` provoquera une erreur de compilation. Par contre, `p` peut-être modifié; par exemple `p = &v` ou `p = p++`

Pointeurs et const

Soit : `int n = 77;`

Quelle sont les différences entre ?

```
a) const int *p = &n;
b) int const *p = &n;
c) int *const p = &n;
d) const int *const p = &n;
```

a. la valeur pointée par `p`, `*p` est constante et ne peut être modifiée; donc `*p = 10` provoquera une erreur de compilation. Par contre, `p` peut-être modifié; par exemple `p = &v` ou `p = p++`

b. une autre manière d'écrire a) et donc équivalent

Pointeurs et const

Soit : `int n = 77;`

Quelle sont les différences entre ?

```
a) const int *p = &n;
b) int const *p = &n;
c) int *const p = &n;
d) const int *const p = &n;
```

a. la valeur pointée par `p`, `*p` est constante et ne peut être modifiée; donc `*p = 10` provoquera une erreur de compilation. Par contre, `p` peut-être modifié; par exemple `p = &v` ou `p = p++`

b. une autre manière d'écrire a) et donc équivalent

c. le pointeur `p` est constant et ne peut être modifé, donc `p = &v` provoquera une erreur de compilation. Par contre, la valeur pointée par `p` peut être modifiée; exemple : `*p = 10` est valide

Pointeurs et const

Soit : `int n = 77;`

Quelle sont les différences entre ?

```
a) const int *p = &n;
b) int const *p = &n;
c) int *const p = &n;
d) const int *const p = &n;
```

a. la valeur pointée par `p`, `*p` est constante et ne peut être modifiée; donc `*p = 10` provoquera une erreur de compilation. Par contre, `p` peut-être modifié; par exemple `p = &v` ou `p = p++`

b. une autre manière d'écrire a) et donc équivalent

c. le pointeur `p` est constant et ne peut être modifé, donc `p = &v` provoquera une erreur de compilation. Par contre, la valeur pointée par `p` peut être modifiée; exemple : `*p = 10` est valide

d. le pointeur `p` ainsi que la valeur pointée par `p`, `*p` sont tous deux constants ! Donc, ni l'un ni l'autre ne peuvent être modifiés