Organisation et utilisation des systèmes d'exploitation 2

2. Rappels sur le langage C

Guillaume Pierre

Université de Rennes 1

Hiver 2014

Table of Contents

Table of Contents

Organisation d'un programme

Un programme Java c'est:

- Plusieurs classes, une classe par fichier.
- Une méthode `main` dans l'un des classes.
- Librairies de classes externes (fichiers jar).

Un programme C c'est:

- Plusieurs fonctions dans n'importe quel nombre de fichiers.
- Une fonction `main` dans l'un des fichiers
- Fichiers "headers"
- Librairies de fonctions externes et leurs headers.

Exemple

```
/* Ceci est un commentaire */

#include <stdio.h>           /* stdio.h est un fichier de header standard */

int main() {                 /* cette fonction est appellée en premier */

  printf("Hello, world!\n"); /* \n est le caractère 'newline' */

  return 0;                  /* main() retourne 0 pour terminer le programme */

}
```


Compiler et exécuter votre programme

- Compiler le programme:

```
gcc -Wall hello.c
```

  - Cela crée un fichier exécutable appelé `a.out`
  - '`-Wall`' signifie "merci d'afficher les warnings lors de la compilation"
    - ☞ Il faut toujours utiliser cette option!
    - ☞ ...**et lire les warnings**! (même s'ils sont parfois difficiles à comprendre)

- Exécuter le programme:

```
$ a.out
Hello, world!
$
```

- Pour donner un autre nom au programme exécutable:

```
gcc -Wall -o mon_programme hello.c
```


Table of Contents

Types de données de base

- Types de données:
  - `char` est un caractère

```
char x='a', y='\n', z='\0';
```

    - ★ On définit les caractères entre apostrophes ('a')
    - ★ Attendion: les double quotes ("a") représentent autre chose
  - `short`, `int`, `long`

```
int a=3, b=-2; long c=1234567890;
```

  - `unsigned short`, `unsigned int` and `unsigned long` sont les versions non signées de `short`, `int` and `long`.
  - `float` and `double` sont des nombres flotaux simple et double précision:

```
float d=3.14159, e=-3.01; f=6.022e23;
```

- Avez-vous remarqué? **Il n'y a pas de type "chaîne de caractères"...**

Tableaux

- Pour définir un tableau:

```
int a[3];   /* a est un tableau de 3 entiers */
```

- Les indices d'un tableau commencent à 0:

```
a[0] = 2; a[1] = 4; a[2] = a[0] + a[1];
int x = a[a[0]];     /* Quelle est la valeur de x? */
```

  - **Attention:** dans cet exemple `a[3]` n'existe pas, mais votre compilateur ne dira rien si vous l'utilisez!
    - ★ Mais votre programme risque de faire des choses étranges...

- Tableaux multidimensionnels:

```
int matrix[3][2];
matrix[0][1] = 42;
```


Chaînes de caractères

- Une chaîne de caractères est un fait un **tableau de caractères**

```
char hello[15]="Hello, world!\n";
```

- Il faut choisir la taille du tableau par rapport à ce qu'on compte stocker dedans
  - On ne peut plus changer la taille après coup
- La fin d'une chaîne de caractères est indiquée par le caractère '\0'
  - Par exemple, `"Hello"` est une chaîne de **6** charactères:

H	e	l	l	o	\0

Manipuler des tableaux

- On ne peut pas copier un tableau dans un autre directement
  - Il faut copier chaque élément un par un

```
int a[3] = {12,24,36};
int b[3];

b = a;     /* Ca ne va PAS marcher!! */

b[0]=a[0];
b[1]=a[1];
b[2]=a[2]; /* Ca va marcher */
```


Manipuler des chaînes de caractères [1/2]

- Il existe des fonctions standard pour manipuler des chaînes de caractères:
  - `strcpy(destination, source)` copie la chaîne **source** dans la chaîne **destination**:

```
char a[15] = "Hello, world!\n";
char b[15];
strcpy(b,a);
```

☞ Attention: `strcpy` **ne vérifie pas** que la chaîne `destination` est assez grande pour stocker la chaîne `source`.

```
char c[10];
strcpy(c,a);  /* GROS GROS PROBLÈME! */
```

Manipulating strings [2/2]

- A la place de `strcpy` il **faut toujours utiliser** `strncpy`:
  - `strncpy` prend un paramètre supplémentaire qui indique le nombre de caractères maximum à copier:

```
char a[15] = "Hello, world!";
char c[10];
strncpy(c,a,10);
c[9] = '\0';      /* Ne pas oublier de rajouter un '\0' à la fin du tableau! */
```


Structures

- On peut créer des nouveaux types de données sous forme de structures:

```
struct Complex {
  float real;
  float imag;
};

struct Complex number;
number.real = 3.2;
number.imag = -2;

struct Parameter {
  struct Complex number;
  char description[32];
};

struct Parameter p;
p.number.real = 42;
p.number.imag = 12.3;
strncpy(p.description, "Un numéro", 31);
p.description[31] = '\0';
```


Les variables

- C a deux sortes de variables:
  - Locales (déclarées à l'intérieur d'une function)
  - Global (déclarées à l'extérieur d'une fonction)

```
int globale;

void f() {
  int locale;

}
```


Les variables locales

- On peut les utiliser uniquement à l'intérieur de la fonction

```
/* ok */
void func1() {
    int a, b;
    a = 0;
    b = 1;
}
```

```
/* ok */
void func2() {
    int a = 0;
    int b = 1;
}
```

```
/* faux! */
void func3() {
    int a = 0;
    a++;
    int b = 1;
}
```

Les variables globales

- Les variables globales sont utilisables dans l'ensemble du fichier où elles sont déclarées
  - Dans n'importe quelle fonction du fichier

```
int i = 7;   /* i est une variable globale */

void foo() {
  printf("i=%d\n", i);
}

int main() {
    printf("i=%d\n", i);
    foo();
}
```


printf()

- **printf** est la fonction pour afficher des messages à l'écran
  - Il existe des variantes (**fprintf**, **sprintf**, etc.)
- printf() est déclarée dans le header standard `stdio.h`
  - ☞ Il faut rajouter "`#include <stdio.h>`" avant d'utiliser printf()
- Les arguments de printf():
  - Une "chaîne de format"
  - Des variables
- Exemple:

```
printf("Ceci est un int: %d\n et ceci est une chaîne: %s\n", 12, "mystring");
```

- printf() va **formater** la chaîne:
  - Replacer '%d' par la première variable: '12'
  - Replacer '%s' par la seconde variable: "mystring"

Exemples de printf()

```
#include <stdio.h>                      |
                                        |
int main(void) {                        |
   int val = 5;                         |
   char c = 'a';                        |
   char str[] = "world";                |
                                        |
   printf("Hello world\n");             |  Hello world
   printf("Hello %d World\n", val);     |  Hello 5 World
   printf("%d %c World\n", val, c);     |  5 a World
   printf("Hello %s\n", str);           |  Hello world
   printf("Hello %d\n", str);           |  ** erreur! **
   return 0;                            |
}                                       |
```

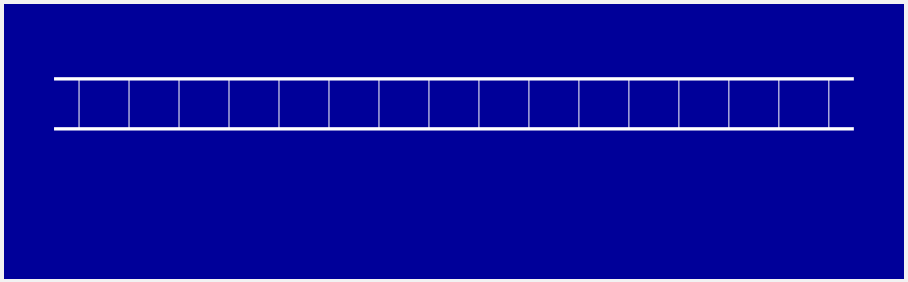

Les chaînes de format

%d	signed int
%u	unsigned int
%x	hexadecimal unsigned int
%c	character
%f	double ou float
%s	string
%%	pour afficher un %

Table of Contents

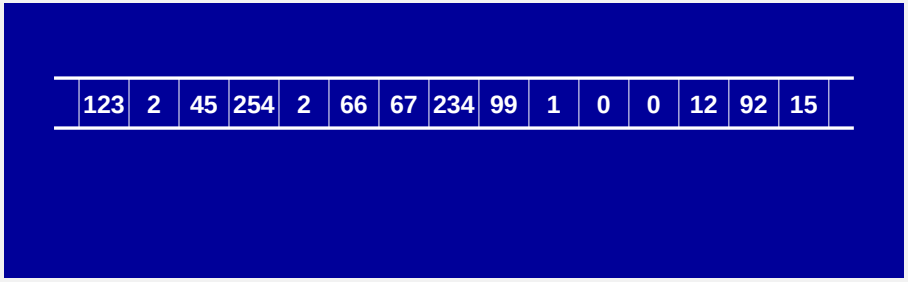
La mémoire en C [1/3]

- Pour un programme C, la mémoire est une suite d'octets...



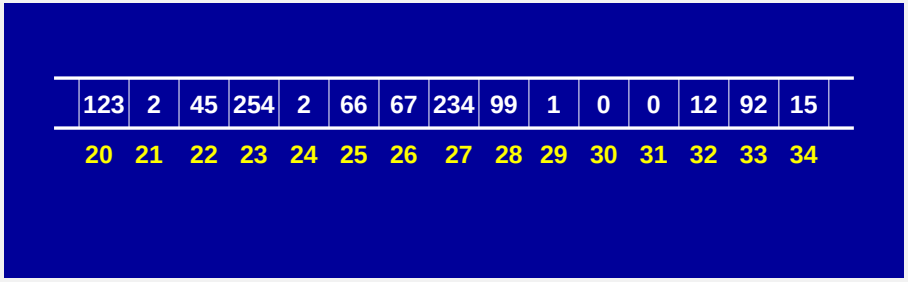
La mémoire en C [1/3]

- Pour un programme C, la mémoire est une suite d'octets...
- Chaque octet a une **valeur**



La mémoire en C [1/3]

- Pour un programme C, la mémoire est une suite d'octets...
- Chaque octet a une **valeur** et une **adresse**



La mémoire en C [2/3] [2/3]

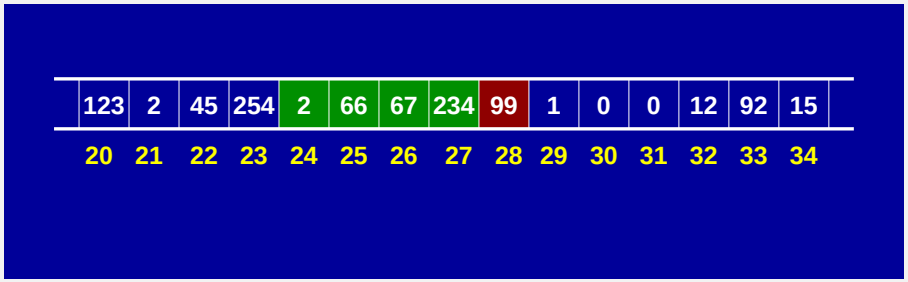
- Quand vous définissez des variables:

```
int count;
unsigned char c;
```

- . . . deux choses se passent:

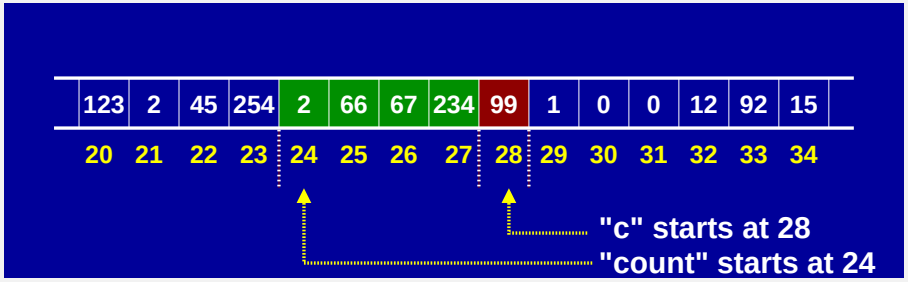
La mémoire en C [3/3]

- De la mémoire est réservée pour stocker les variables. . .



La mémoire en C [3/3]

- De la mémoire est réservée pour stocker les variables. . .
- . . . et le compilateur mémorise les **adresses des variables** qu'il a créé



Les variables

- Chaque variable a donc deux propriétés importantes:
  1. La **valeur** stockée dans la variable
     - ☞ Quand vous indiquez le nom d'une variable, vous accédez à sa valeur
  2. L'**adresse** en mémoire où la variable est placée
     - ★ Similaire à une référence en Java (mais pas complètement identique)

Pointeurs

Une variable qui contient **l'adresse d'une autre variable** s'appelle un **pointeur**

Table of Contents

Les pointeurs

- **Un pointeur est une variable qui contient une adresse mémoire**
- On déclare un pointeur avec le signe *****

```
int *ptr;                /* Pointeur vers un int */
unsigned char *ch;       /* Pointeur vers un unsigned char */
struct ComplexNumber *c; /* Pointeur vers un struct ComplexNumber */
int **pp;                /* Pointeur vers un pointer vers un int */
void *v;                 /* Pointeur vers n'importe quoi (dangereux!) */
```


Définir un pointeur

- Pour utiliser un pointeur il faut d'abord lui donner une valeur
  - Comme n'importe quelle variable
- Le signe '&' indique **l'adresse mémoire** d'une variable

```
int i = 8;

int *p;          /* p est un pointer vers un int */

p = &i;          /* p contient l'adresse de la variable i */

double *d = &i; /* ERREUR, mauvais type de pointeur */
```

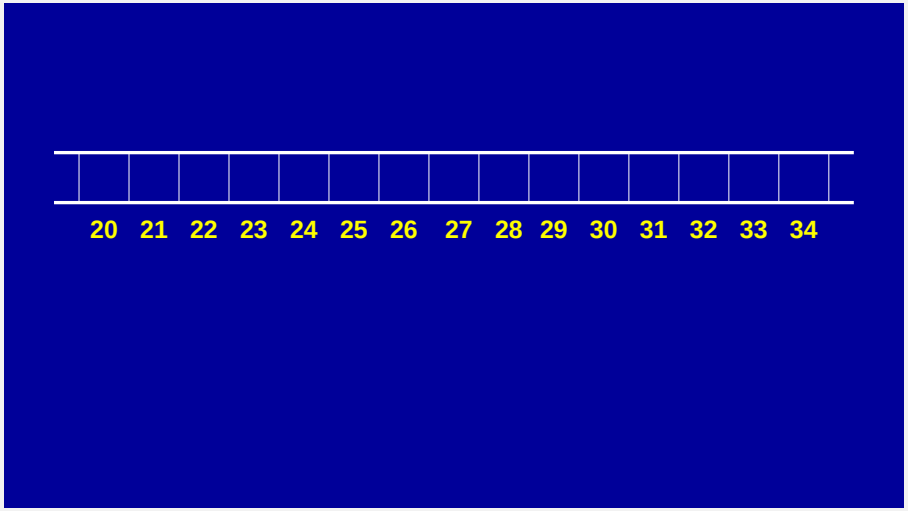

Utiliser un pointeur

- Quand vous avez un pointeur vous pouvez accéder à la valeur stockée à l'adresse pointée avec le caractère '*'

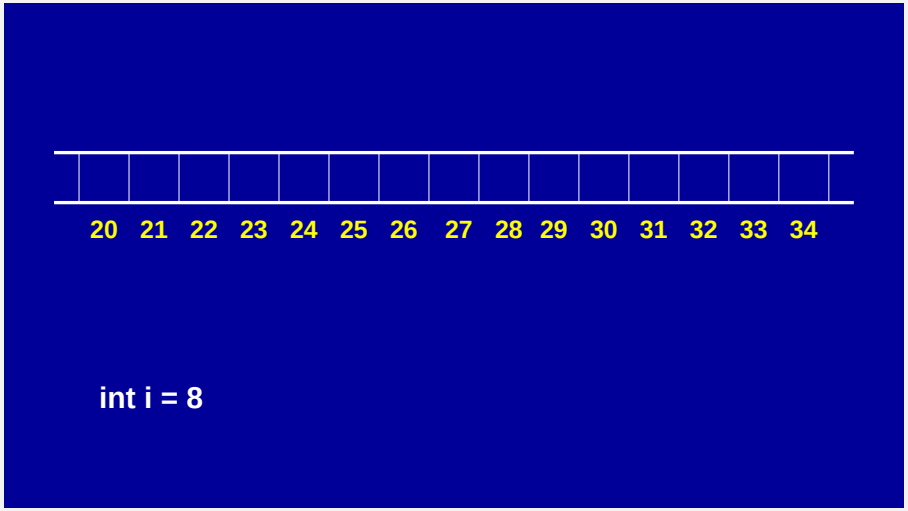
```
int i = 8;
int *p = &i;      /* p pointe vers i */
int j = *p;       /* j contient maintenant 8 */
*p = 12;          /* i contient maintenant 12 */
```

☞ **Attention**, le signe '*' sert à deux usages différents:
  - Pour **declarer** une variable de type pointeur: `int *p;`
  - Pour **déréferencer** un pointer: `*p=12;`

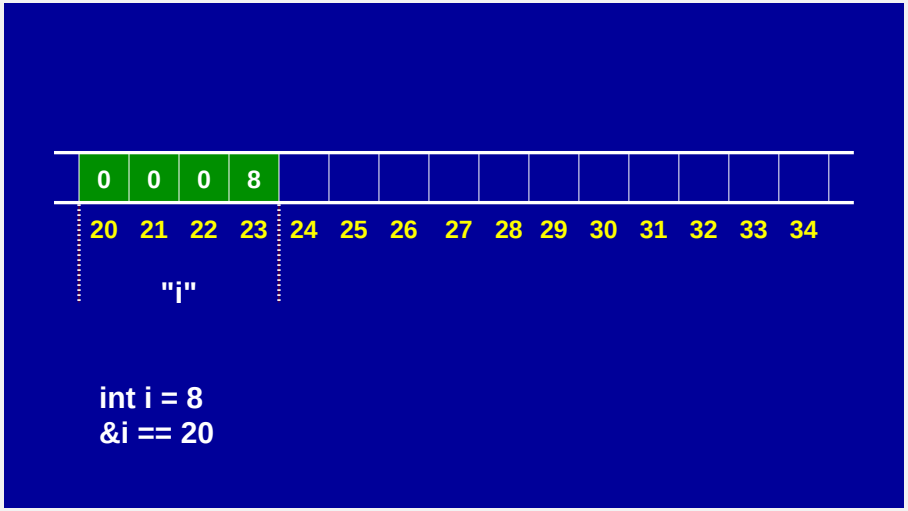
Pointeurs



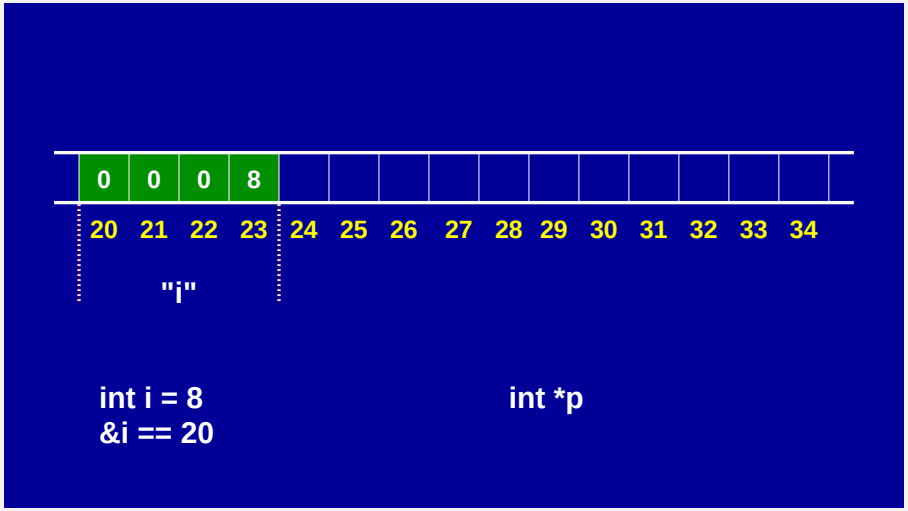
Pointeurs



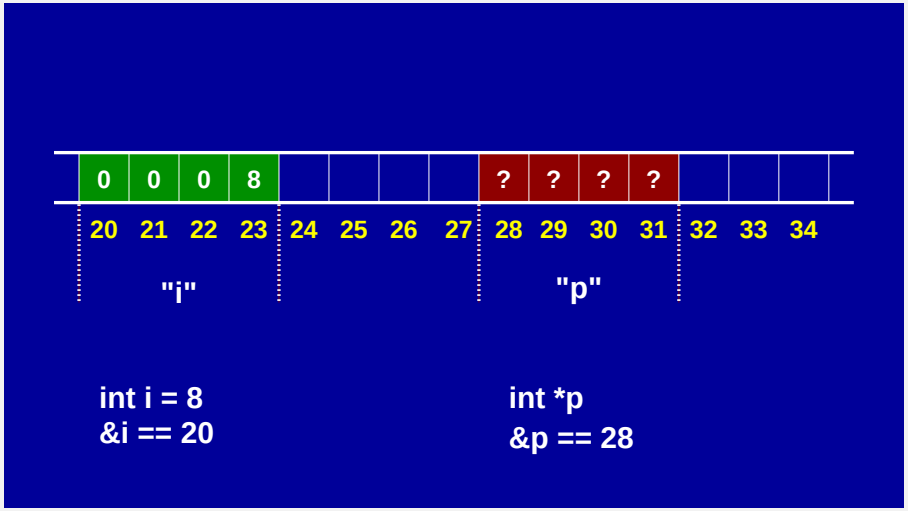
Pointeurs



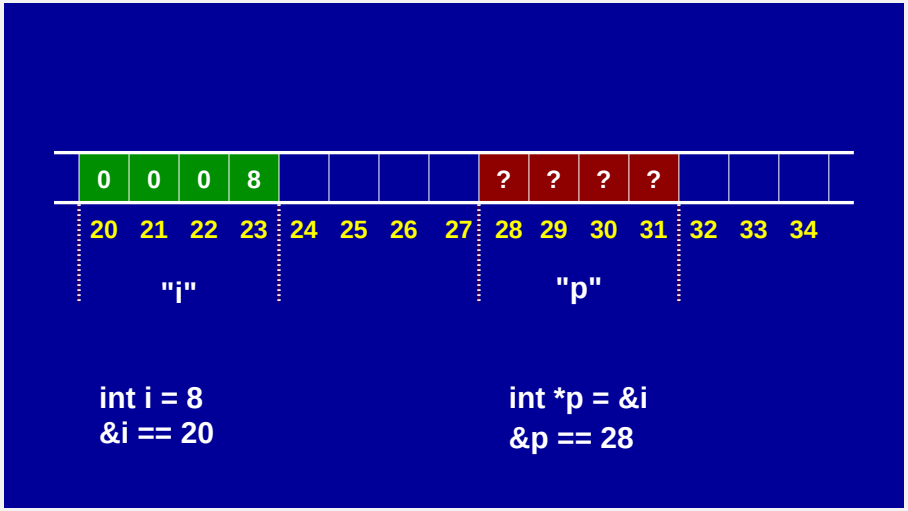
Pointeurs



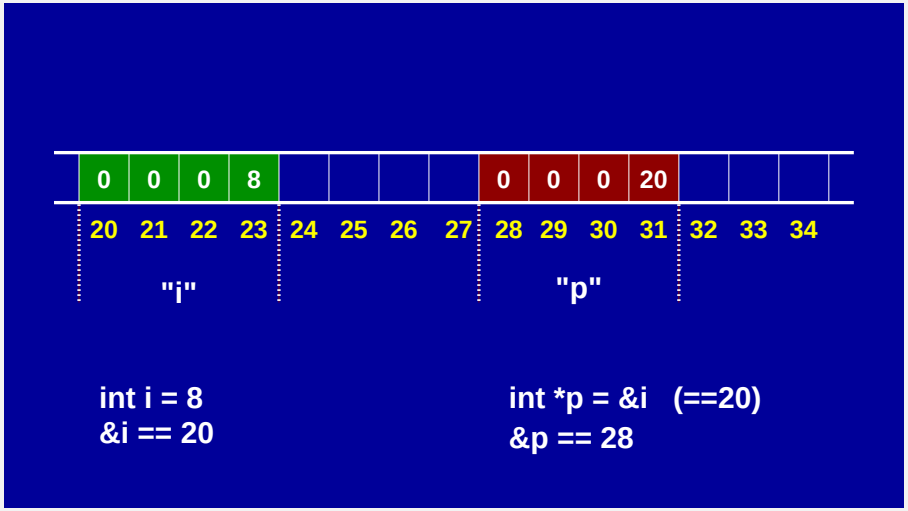
Pointeurs



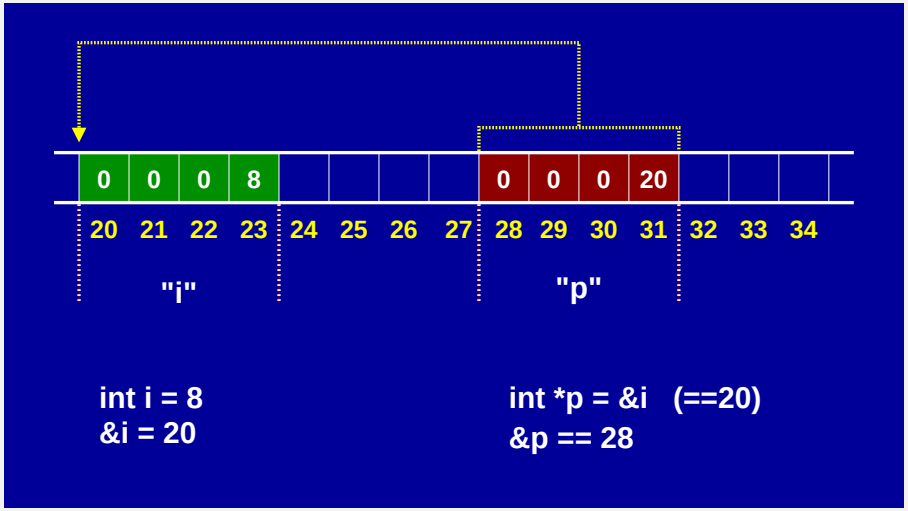
Pointeurs



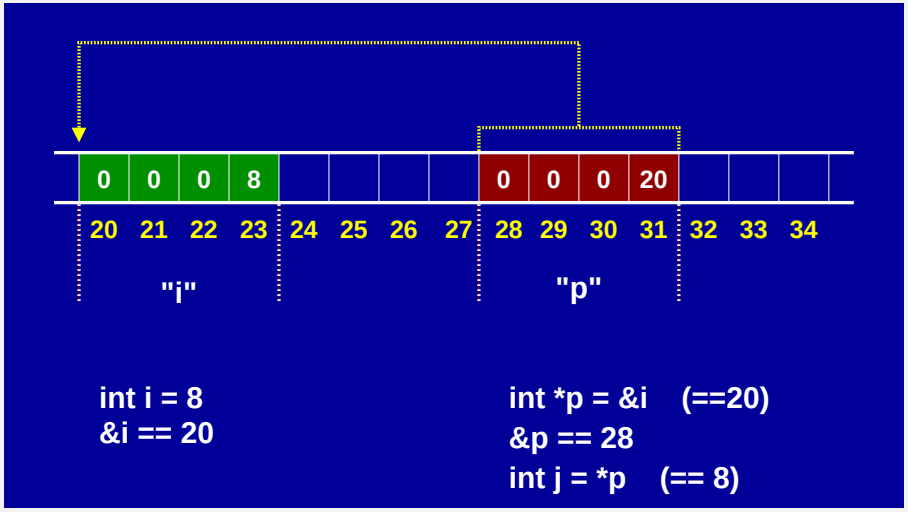
Pointeurs



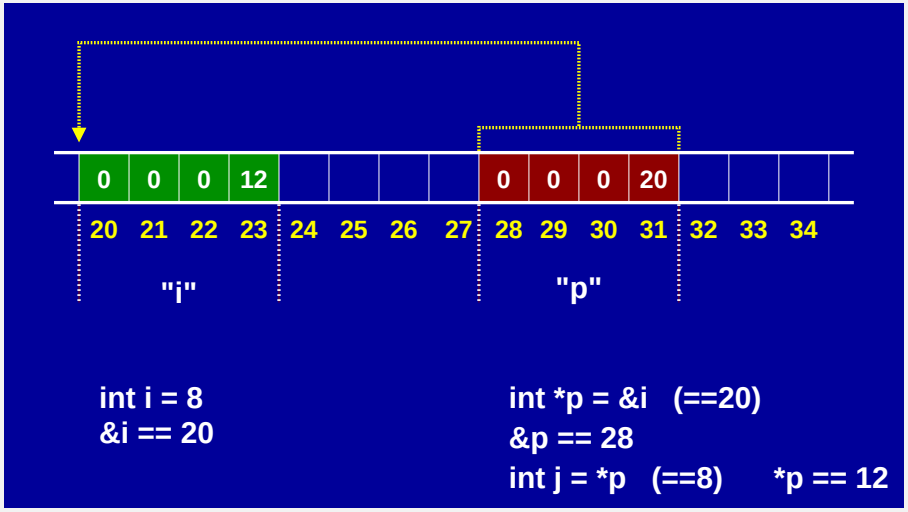
Pointeurs



Pointeurs



Pointeurs



Erreur classique avec les pointeurs

- Quand vous créez un pointeur vous ne connaissez pas sa valeur initiale
  - ☞ Il faut d'abord donner une valeur au pointeur avant de pouvoir l'utiliser

```
int main() {
  int *i;      /* On ne sait pas où pointe le pointeur */
  int j = *i;  /* FAUX! */
}
```

- Ce programme accède à une adresse mémoire bizarre
  - Très probablement votre programme n'a pas **l'autorisation** d'accéder à cette adresse
  - Résultat:

```
$ a.out
Segmentation fault (core dumped)
$
```

(Nous étudierons le "core dumped" dans un cours suivant...)

Table of Contents

Passer des paramètres par référence [1/2]

- En C **tous les paramètres sont passés par valeur**
  - On crée une copie temporaire du paramètre
  - La fonction accède à la copie mais jamais à l'original

```
#include <stdio.h>

void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int x = 9;
  int y = 5;
  swap(x, y);
  printf("x=%d y=%d\n", x, y); /* Résultat: x=9 y=5 */
  return 0;
}
```


Passer des paramètres par référence [2/2]

- Les pointeurs permettent de passer des paramètres par référence
  - On crée une copie temporaire du pointeur
  - Cette copie pointe au même endroit que le pointeur original

```
#include <stdio.h>

void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int x = 9;
  int y = 5;
  swap(&x, &y);
  printf("x=%d y=%d\n", x, y); /* Résultat: x=5 y=9 */
  return 0;
}
```


Passer une structure en paramètre [1/2]

- Les structures sont également passées par valeur

```
#include <stdio.h>
struct data {
   int counter;
   double value;
};
void add(struct data d, double value) {
   d.counter++;
   d.value += value;
}
int main() {
  struct data d = { 0, 0.0 };
  add(d, 1.0);
  add(d, 3.0);
  printf("counter=%d, value=%f\n", d.counter, d.value); /* counter=0, value=0.0 */
  return 0;
}
```


Passer une structure en paramètr [2/2]

- Pour corriger ce programme il faut passer la structure par référence:

```
#include <stdio.h>
struct data {
   int counter;
   double value;
};
void add(struct data *d, double value) {
   (*d).counter++;
   (*d).value += value;
}
int main() {
  struct data d = { 0, 0.0 };
  add(&d, 1.0);
  add(&d, 3.0);
  printf("counter=%d, value=%f\n", d.counter, d.value); /* counter=2, value=4.0 */
  return 0;
}
```

Une abréviation utile

- On utilise souvent des syntaxes comme ceci:

```
(*pointeur).champ = valeur;
```

- Il existe une autre notation strictement équivalente:

```
pointeur->champ = valeur;
```

- Par exemple:

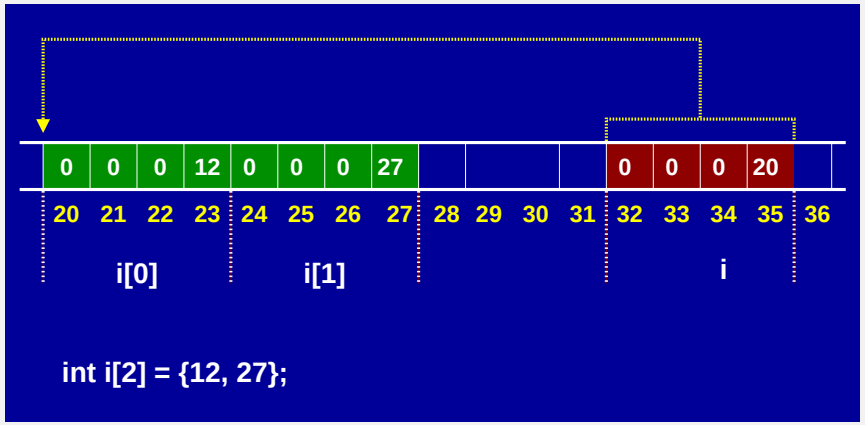
```
struct data {
   int counter;
   double value;
};

void add(struct data *d, double value) {
   d->counter++;
   d->value += value;
}
```


Table of Contents

Tableaux et pointeurs [1/4]

- Que se passe-t-il quand vous créez un tableau `int i[2]`:
  - Le compilateur réserve suffisament de mémoire pour stocker deux int
  - Il conserve **l'adresse** du début du tableau dans une variable de type pointeur



Tableaux et pointeurs [2/4]

- On peut utiliser le pointeur à la place du tableau

```
#include <stdio.h>

void func1(int p[], int size) { }

void func2(int *p, int size) { }

int main() {
  int array[5];
  func1(array, 5);
  func2(array, 5);
  return 0;
}
```

Tableaux et pointeurs [3/4]

- On peut même utiliser des indices avec les pointeurs (comme pour des tableaux)!

```
void clear(int *p, int size) {
    int i;
    for (i=0;i<size;i++) {
         p[i] = 0;
    }
}

int main() {
  int array[5];
  clear(array, 5);
  return 0;
}
```


Tableaux et pointeurs [4/4]

- Une chaîne de caractères est en fait un **tableau** de caractères:

```
int main() {
   char s1[32] = "Hello, world!\n";
   char *s2;
   char s3[32];
   s2 = s1;           /* s1 and s2 pointent vers le same tableau de caractères */
   strncpy(s3,s1,32); /* s3 contient une copie de s1 */
   s3[31] = '\0';
```


Récupérer des arguments en ligne de commande [1/2]

- La fonction `main()` peut prendre des arguments
  - Afin de récupérer les arguments de la ligne de commande passés par l'utilisateur

```
int main(int argc, char** argv) {
   ...
}
```

  - `argc` contient le **nombre** d'arguments de la ligne de commande
    - ★ Y compris le nom du programme
  - `argv` est un **tableau** de **chaînes de caractères** qui contient chaque paramètre (y compris le nom du programme)

Récupérer des arguments en ligne de commande [2/2]

- Exemple:

```
int main(int argc, char **argv) {
  int i;
  printf("  Il y a %d argument(s):\n", argc);
  for (i=0;i<argc;i++) {
    printf("  Argument %d: %s\n", i, argv[i]);
  }
}
```

- Essayons:

```
$ a.out
  Il y a 1 argument(s):
  Argument 0: a.out
$ ./a.out foo bar
  Il y a 3 argument(s):
  Argument 0: ./a.out
  Argument 1: foo
  Argument 2: bar
$
```

Table of Contents

Arithmétique de pointeurs [1/2]

- Les pointeurs sont des variables (presque) comme les autres
- On peut effectuer des **calculs** sur les pointeurs
  - On peut utiliser +, -, ++, - sur les pointeurs
  - Il n'existe pas d'équivalent en Java
- Attention, les operateurs utilisent la taille des types de variables!

```
int i = 8;
int *p = &i;
p++;  /* augmente p de sizeof(int) */

char *c;
c++; /* augmente c de sizeof(char) */
```

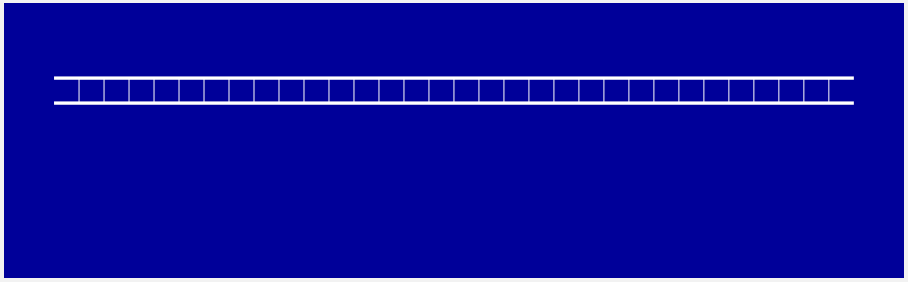

Arithmétique de pointeurs [2/2]

- On utilise l'arithmétique de pointeurs quand on manipule des tableaux:

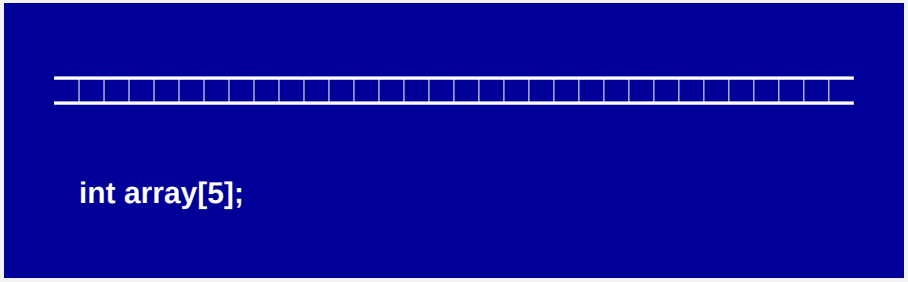
```
int i;
int tableau[5];
int *p = tableau;

for (i=0;i<5;i++) {
   *p = 0;
   p++;
}
```


Exemple

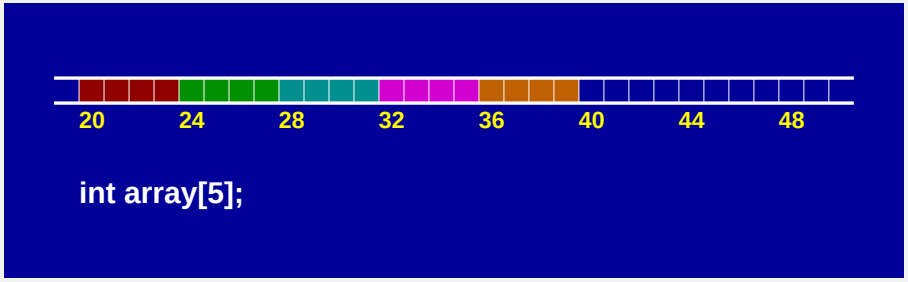


Exemple



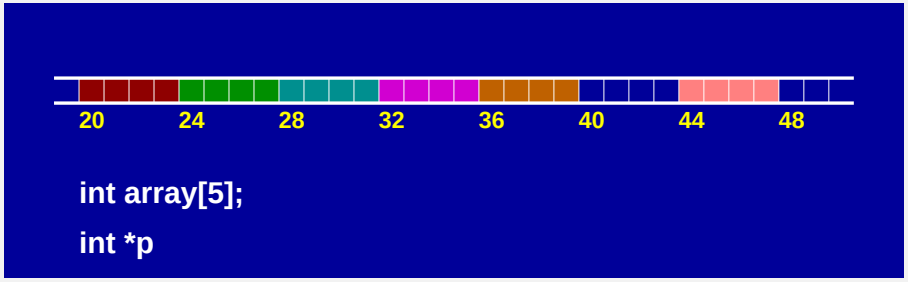
```
int array[5];
```


Exemple



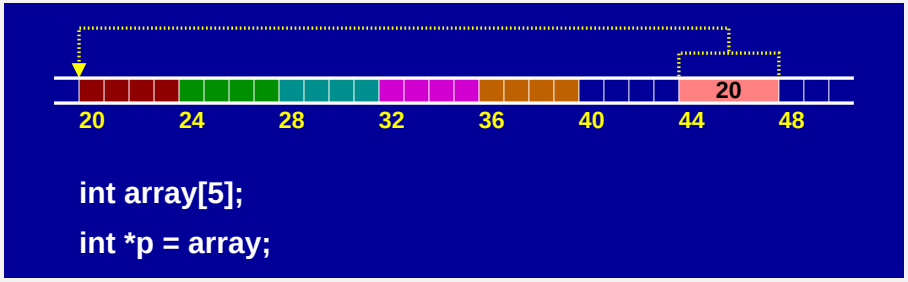

```
int array[5];
```


Exemple



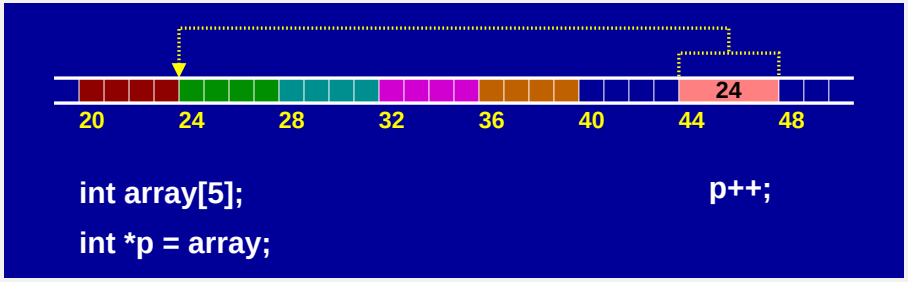

```
int array[5];
int *p
```


Exemple

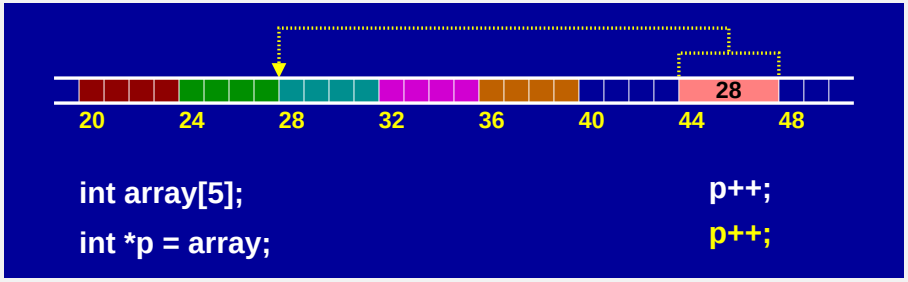



```
int array[5];
int *p = array;
```

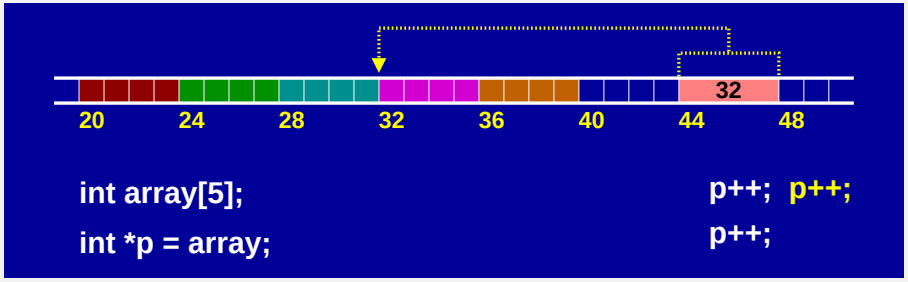
Exemple



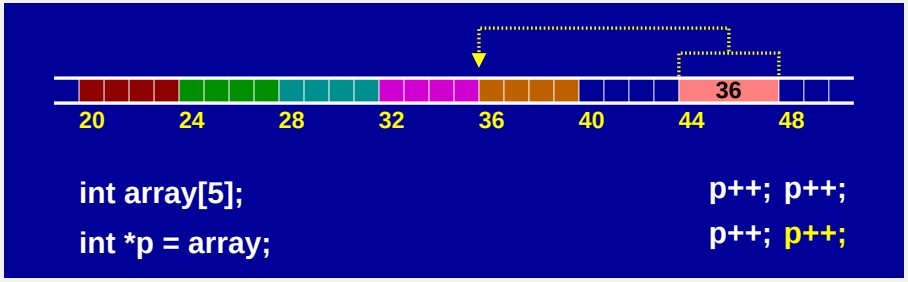
Exemple



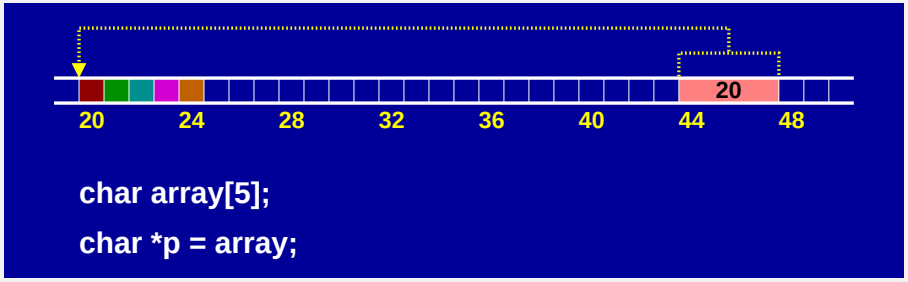
Exemple



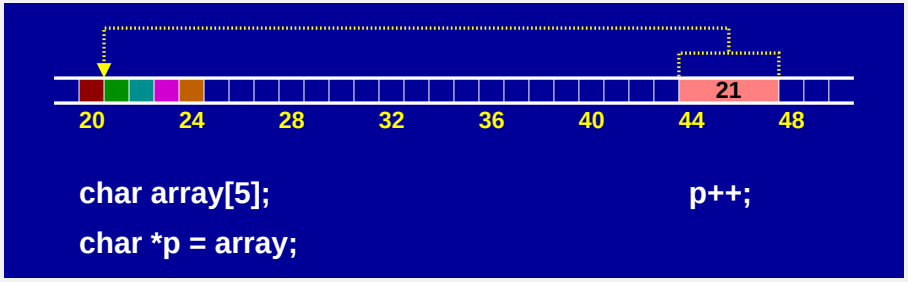
Exemple



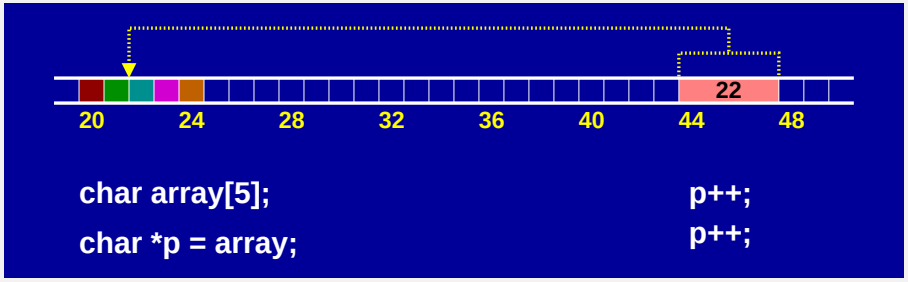
Exemple



Exemple



Exemple



Exemple