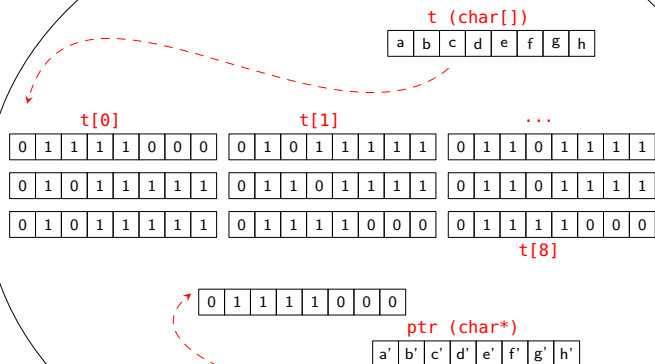


Algo/Prog – DUT 1

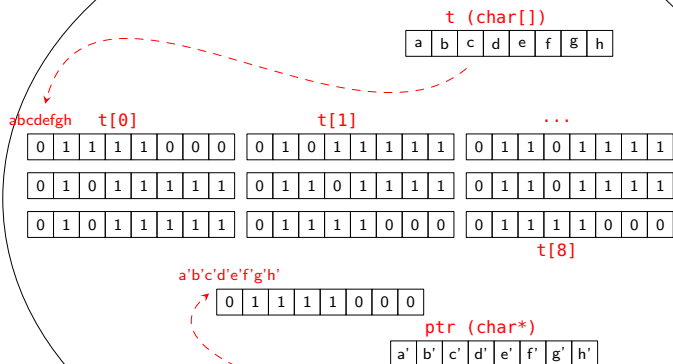
- ① Introduction
- ② Premiers programmes
- ① Définition de procédures et fonctions
- ② Conditionnelle (vu en TP)
- ① Instructions complexe (boucles)
- ① Variables, types
- ② Représentation des données en mémoire
- ① Pointeurs, etc

- 1 Pointeurs, etc
 - Pointeurs statiques
 - Fonctions modifiant leurs arguments
 - `scanf`, `printf`
 - Allocation dynamique
 - `struct`
 - Directives du préprocesseur `cpp`

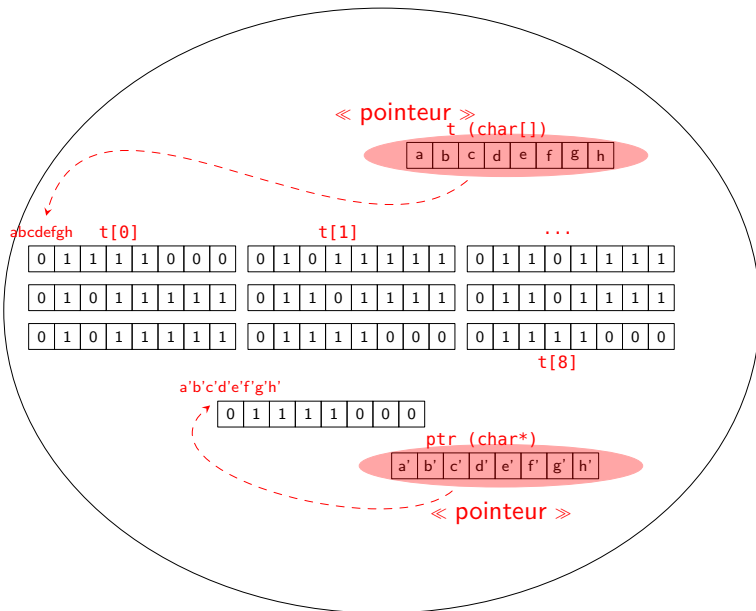
Pointeurs en mémoire



Pointeurs en mémoire



Pointeurs en mémoire



Pointeur = adresse mémoire

- pointeur = variable contenant une adresse
- en C :
 - ▶ tableaux = pointeurs EN MÉMOIRE
 - ▶ tableaux \neq pointeurs DANS LE LANGAGE

Déclarer et utiliser un pointeur

```
int * ptr;
```


Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;
```

Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;  
ptr = &x;
```

Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;  
ptr = &x;  opérateur « adresse de »
```

Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;  
ptr = &x;  
ecrireInt(x);      ~→ 8  
ecrireInt(*ptr);   ~→ 8
```

Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;  
ptr = &x;  
ecrireInt(x);      ~→ 8  
ecrireInt(*ptr);   ~→ 8   opérateur « contenu de »
```

Déclarer et utiliser un pointeur

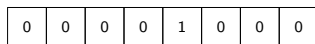
```
int * ptr;  
int x= 8;  
ptr = &x;  
ecrireInt(x);      ~> 8  
ecrireInt(*ptr);   ~> 8  
  
x = 9;  
ecrireInt(x);      ~> 9  
ecrireInt(*ptr);   ~> 9
```

Déclarer et utiliser un pointeur

```
int * ptr;  
int x= 8;  
ptr = &x;  
ecrireInt(x);      ~> 8  
ecrireInt(*ptr);   ~> 8  
  
x = 9;  
ecrireInt(x);      ~> 9  
ecrireInt(*ptr);   ~> 9  
  
*ptr = 10;  
ecrireInt(x);      ~> 10  
ecrireInt(*ptr);   ~> 10
```

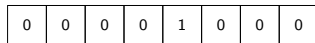
Pointeur en argument d'une fonction

```
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x);  
}
```



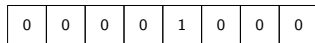
Pointeur en argument d'une fonction

```
void p(int * a) {  
    *a = 0;  
}  
  
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x);  
}
```



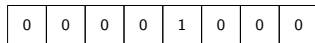
Pointeur en argument d'une fonction

```
void p(int * a) {  
    *a = 0;  
}  
  
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x);  
}
```



Pointeur en argument d'une fonction

```
void p(int * a) {  
    *a = 0;  
}  
  
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x);  
}
```

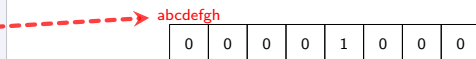


Pointeur en argument d'une fonction

```
void p(int * a) {  
    *a = 0;  
}
```

```
int main() {  
    int x ← 8;  
    ecrireInt(x); ~ 8  
    p(&x);  
}
```

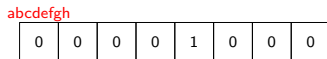
abcdefgh



0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

Pointeur en argument d'une fonction

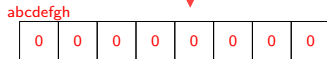
```
void p(int * a) {  
    *a = 0;  
}  
  
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x); p(abcdefgh);  
}
```



Pointeur en argument d'une fonction

```
void p(int * a) {  
    *a = 0;    *(abcdefgh) = 0;  
}
```

```
int main() {  
    int x= 8;  
    ecrireInt(x); ~ 8  
    p(&x);    p(abcdefgh);  
    ecrireInt(x); ~ 0  
}
```



scanf

- scanf permet de lire des valeurs au clavier

```
int n,m,p;  
scanf("%d",&n); // lit un entier au clavier et modifie la variable n  
scanf("%d,%d",&m,&p); // deux entiers séparés par ',',' modifie m et p  
double f1, f2;  
scanf("%lf:%lf",&f1,&f2); // deux doubles séparés par ':'
```



"%s" obsolète (buffer overflows)

printf

- `printf` : même codes de format que `scanf` pour écrire à l'écran

```
int n=10,m=20;
printf("Valeurs entières: %d et %d.",n,m);
  ~→ Valeurs entières: 10 et 20.
double f=8.9086;
printf("Valeurs: %d, %lf et %lf.",7,f,(double)8.9);
```


Des pointeurs sans variable associée

- `x = &p;` partage d'une cellule mémoire entre deux variables
- pointeurs non limité à cet usage
- On peut « allouer » (réserver) de la mémoire pour un pointeur
- `malloc`
- `free`

malloc


```
char * ptr;  
ptr = (char *) malloc(1);
```

malloc

- prend en argument la taille de la zone mémoire demandée

```
char * ptr;  
ptr = (char *) malloc(1);
```

malloc

- prend en argument la taille de la zone mémoire demandée
- retourne l'adresse de la zone mémoire allouée sans type (`void *`)
- cast pour forcer le type 

```
char * ptr;  
ptr = (char *) malloc(1);
```

malloc

- prend en argument la taille de la zone mémoire demandée
- retourne l'adresse de la zone mémoire allouée sans type (`void *`)
- cast pour forcer le type 
- calcul de la taille partiellement confié au compilateur


```
char * ptr;  
ptr = (char *) malloc(1);  
int * ptr2;  
ptr2 = (int *) malloc(sizeof(int));
```

malloc

- prend en argument la taille de la zone mémoire demandée
- retourne l'adresse de la zone mémoire allouée sans type (`void *`)
- cast pour forcer le type 
- calcul de la taille partiellement confié au compilateur
- Tableaux alloué dynamiquement


```
char * ptr;  
ptr = (char *) malloc(1);  
int * ptr2;  
ptr2 = (int *) malloc(sizeof(int));  
int * ptrtab;  
ptrtab = (int *) malloc(12*sizeof(int));
```

malloc

- prend en argument la taille de la zone mémoire demandée
- retourne l'adresse de la zone mémoire allouée sans type (`void *`)
- cast pour forcer le type 
- calcul de la taille partiellement confié au compilateur
- Tableaux alloué dynamiquement
- retourne 0 si échec d'allocation

```
char * ptr;
ptr = (char *) malloc(1);
int * ptr2;
ptr2 = (int *) malloc(sizeof(int));
int * ptrtab;
ptrtab = (int *) malloc(12*sizeof(int));
if (ptrtab) { ... } else { exit(1); }
```

free

- `malloc` : pas de variable associée à la zone mémoire
- non détruit à la sortie de la fonction! 
- très utile mais penser à libérer (fuite mémoire)
- `free(addr)`

```
char * ptr;
ptr = (char *) malloc(1);
int * ptr2;
ptr2 = (int *) malloc(sizeof(int));
int * ptrtab;
ptrtab = (int *) malloc(12*sizeof(int));
if (ptrtab) { ... } else { exit(1); }
...
free(ptr); .... free(ptr2); ... free(ptrtab);
```


Pointeurs

```
#include<stdlib.h> // Pour malloc
#include" inout.h"
void afficheTab(int * tab, int n) {
    int i;
    for (i=0 ; i<n ; i++){ printf("%d, ",tab[i]); }
}

int main() {
    ecrireString("Combien de cases?");
    int i, n = lireInt();
    int *t = (int*) malloc(n*sizeof(int));
    if (!t) { exit(1); } // Sortie du programme
    afficheTab(t,n);
    ecrireSautDeLigne();
    for (i=0;i<n;i++){ t[i] = i; }
    afficheTab(t,n);
}
```

Structure de données complexes

Comment relier plusieurs données de différents types (\neq tableau) ?

Ex : nom + num sécurité sociale + solde.

- solution 1

- ▶ plusieurs tableaux
- ▶ association = même indice dans le tableaux
- ▶ ex : tableau noms + tableau numéro S.S. + tableau soldes.

- solution 2

- ▶ un seul tableau
- ▶ cellule = « tableau de types différents ».
- ▶ case 1 : nom, case 2 : num S.S., 3 : solde etc.

Tableaux hétérogènes

Tableaux hétérogènes :

- objets dans les langage OO
- records (enregistrements) dans les autres
- en C : `struct`

Principe

- Nombre fixe de case
- Désignées par des noms (~~numéro~~)
- `t.nom` (~~t[nom]~~)

struct – déclaration, utilisation

```
struct { ... } var;
```

struct – déclaration, utilisation

```
struct { ... } var;  
    int    n;  
    char   c;
```

struct – déclaration, utilisation

```
struct { ... } var;  
    int    n;  
    char   c;
```

```
struct {  
    int nom1;  
    char* nom2;  
    double nom3; } x;
```

struct – déclaration, utilisation

```
struct { ... } var;  
    int    n;  
    char   c;
```

type

```
struct {  
    int nom1;  
    char* nom2;  
    double nom3; } x;
```

struct – déclaration, utilisation

```
struct { ... } var;  
    int    n;  
    char   c;
```

```
struct {  
    int nom1;  
    char* nom2; nom de variable  
    double nom3; } x;
```


struct – déclaration, utilisation

```
struct { ... } var;  
    int    n;  
    char   c;
```

```
struct {  
    int nom1;  
    char* nom2;  
    double nom3; } x;  
x.nom1 = 12;  
x.nom2 = "toto";  
x.nom3 = 23.6;  
printf("%d , %s , %lf",x.nom1, x.nom2, x.nom3);
```

struct et typedef

Pour abrégé : synonyme de type :

```
typedef struct { ... } nomtype;
```

struct et typedef

Pour abrégé : synonyme de type :

```
typedef struct { ... } nomtype;
```

```
typedef struct {  
    int nom1;  
    char* nom2;  
    double nom3; } nomtype;
```

```
nomtype y;  
y.nom1 = 12;
```

struct et typedef

Pour abrégé : synonyme de type :

```
typedef struct { ... } nomtype;
```

```
typedef struct {  
    int nom1;  
    char* nom2;  
    double nom3; } nomtype;
```

type

nomtype y;

y.nom1 = 12;

Directives de pré-processing

- Transformation du programme avant la compilation

```
#include<xxx.h>
```

```
#include"xxx.h"
```

```
#define BOOL int
```

Directives de pré-processing

```
#define WINDOWS Changer + recompiler
```

```
#ifdef WINDOWS  
    /* Code source pour Windows */  
#endif
```

```
#ifdef LINUX  
    /* Code source pour Linux */  
#endif
```

```
#ifdef MAC  
    /* Code source pour Mac */  
#endif
```

+ #ifndef WINDOWS

etc