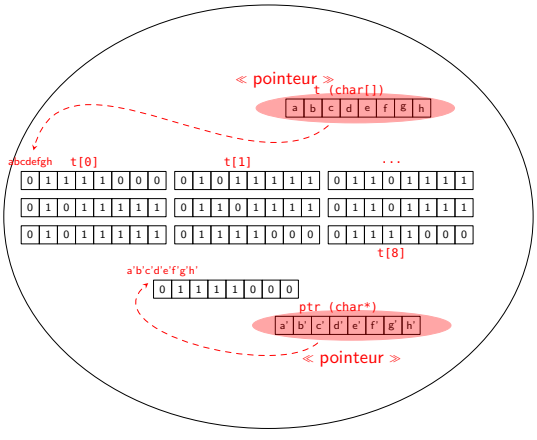
Algo/Prog – DUT 1

Notes

Notes

Notes

Pointeurs en mémoire



Notes

Pointeur = adresse mémoire

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

Notes

Déclarer et utiliser un pointeur

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

x = 9;
ecrireInt(x);       ⇝ 9
ecrireInt(*ptr);    ⇝ 9

*ptr = 10;
ecrireInt(x);       ⇝ 10
ecrireInt(*ptr);    ⇝ 10
```

Notes

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
}
```

Notes

`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)

Notes

`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);
```

Notes

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`

Notes

`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); }
```

Notes

`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);
```

Notes

Pointeurs

```c
#include<stdlib.h> // Pour malloc
#include"inout.h"
void affichTab(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
  affichTab(t,n);
  ecrireSautDeLigne();
  for (i=0;i<n;i++){ t[i] = i; }
  affichTab(t,n);
}
```

Notes

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.

Notes

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]~~)

Notes

`struct` – déclaration, utilisation

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

type

```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);
```

nom de variable

Notes

`struct` et `typedef`

Pour abréger : synonyme de type :

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

```
typedef struct {
  int nom1;
  char* nom2;
  double nom3; } nomtype;
type
nomtype y;
y.nom1 = 12;
```

Notes

Directives de pré-processing

- Transformation du programme avant la compilation

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

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

```
#define BOOL int
```

Notes

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

Notes

Notes