

Techniques de Programmation

Structuration, pointeurs et tableaux

Nicolas Belloir

nicolas.belloir@univ-pau.fr

Structures, unions, enumerations et types

Regroupons les informations de même nature!

- Un **enregistrement**
 - est un mécanisme permettant de regrouper un certain nombre de variables de types différents au sein d'une **même entité**.
 - Les éléments d'un enregistrement sont appelés **les champs**
- En langage C/C++, un enregistrement est appelé une **structure** et un champ est appelé un **membre** de la structure

Struct et enum : vers l'abstraction

```
struct vecteur
{
    float x;
    float y;
};
float norme(const vecteur* v)
{
    return sqrt(v->x*v->x+v->y*v->y);
}
int main()
{
    struct vecteur mon_vecteur={1,2};
    float n=norme(&mon_vecteur);
    printf("%f\n",n);
}
```

```
int main()
{
    float v[2]={1,2};
    float n=sqrt(v[0]*v[0]+v[1]*v[1]);
    printf("%f\n",n);
}
```

Deux codes faisant la même chose!!!

Le struct favorise la compréhension!

Factorisation : favoriser modularité et flexibilité

```
int main()  
{  
    struct vecteur mon_vecteur_1={1,2};  
    struct vecteur mon_vecteur_2={4,5};  
    struct vecteur mon_vecteur_3={5,-1};  
  
    float n1=norme(&mon_vecteur_1);  
    float n2=norme(&mon_vecteur_2);  
    float n3=norme(&mon_vecteur_3);  
}
```



Et si on changeait la norme 2 par une norme infinie????

```
float norme(const vecteur* v)  
{  
    return max(v->x, v->y);  
}
```

```
int main()  
{  
    float v1[2]={1,2};  
    float v2[2]={4,5};  
    float v3[2]={5,-1};  
  
    float n1=sqrt(v1[0]*v1[0]+v1[1]*v1[1]);  
    float n2=sqrt(v2[0]*v2[0]+v2[1]*v2[1]);  
    float n3=sqrt(v3[0]*v3[0]+v3[1]*v3[1]);  
}
```



Aïe aïe aïe ...

Enumérations

- Les énumérations
 - permettent de déclarer des constantes nommées.
 - fonctionnent syntaxiquement comme des structures et peuvent être déclarées sous forme de variable.

```
enum {LUNDI , MARDI , MERCREDI , JEUDI , VENDREDI , SAMEDI , DIMANCHE} ;
```

```
enum jour {LUNDI , MARDI , MERCREDI , JEUDI , VENDREDI , SAMEDI ,  
DIMANCHE} ;
```

```
enum jour j1 , j2 ;
```

```
j 1 = LUNDI ;
```

```
j 2 = MARDI ;
```

Bonnes pratiques: Dénomination

```
enum type_carburant {gasoil, essence, sans_plomb, GPL};  
struct voiture  
{  
    int immatriculation;  
    int kilometrage;  
    enum type_carburant carburant;  
};
```



```
struct voiture v1;  
v1.carburant=sans_plomb;
```

OK: haut niveau

```
struct int_triplet  
{  
    int u_nbr; //immatriculation  
    int u_km;  //kilometrage  
    int u_c;   //carburant (0-gasoil,  
                //          1-essence,  
                //          2-sans_plomb,  
                //          3-GPL)  
};
```



```
struct int_triplet v2;  
v2.u_c=2;
```

mélange: int / voiture
=>niveau d'abstraction non homogène

Structures : attention à l'homogénéité!

```
#define N 50

struct arbre
{
    int hauteur_tronc;
    char couleur_tronc[N];
    int largeur_tronc;
    int nombre_feuille;
    char couleur_feuille[N];
    int profondeur_racines;
};
```

Peu homogène! Mélange
de concepts

```
#define N 50
struct tronc_arbre
{
    int hauteur;
    char couleur[N];
    int largeur;
};
struct feuille_arbre
{
    int nombre;
    char couleur[N];
};
struct racine_arbre
{
    int profondeur;
};
```

```
struct arbre
{
    struct tronc_arbre tronc;
    struct feuille_arbre feuille;
    struct racine_arbre racine;
};
```

Homogène et modulaire

Structures : attention à l'homogénéité!

Trop de paramètres
Mémoire humaine limitée

En moyenne 3-6 paramètres

```
struct arbre
{
    int largeur;
    int hauteur;
    enum type_arbre;
    int nombre_embanchement;
    int circonference;
    int poids;
    int profondeur_sous_sol;
    int nombre_fleurs;
    int flux_seve;
    int mois_fleurissement;
    int nombre_jour_fleurissement;
    int temperature_maximale;
    int quantitee_eau;
    int valeur_marche;
    int resistance_vent;
};
```

Synthèse

- Une **bonne** struct:

- Encapsule des données
- Modélise un objet/une abstraction
- Contient des paramètres homogènes

- Une **mauvaise** struct:

- N'apporte pas d'information
- Contient des informations sans coherences, ne modélise rien
- Ne sert que de conteneur de variables au niveau C
- Contient des noms peu significatifs

Struct et fonctions

- Coupler les 2 permet :
 - une meilleure abstraction
 - Manipulation de données complexes
 - Cacher la complexité
 - Factoriser le code

```
int main()
{
    float x1=0,y1=0,z1=0,x2=4,y2=-5,z2=6;
    float R1=1,R2=2;

    float volume_1=4.0/3*M_PI*R1*R1*R1;
    float volume_2=4.0/3*M_PI*R2*R2*R2;
}
```

pas lisible:

=> abstraction *sphere* n'apparaît pas

- pas d'évolution possible:

=> changer implémentation=
réécrire totalement le code

```
int main()
{
    struct sphere s1; sphere_init(&s1,0,0,0,1.0);
    struct sphere s2; sphere_init(&s2,4,-5,6,2.0);

    float volume_1=sphere_volume(&s1);
    float volume_2=sphere_volume(&s2);
}
```

lisible

+ évolutif

Typage

- Mieux que la structure, déclarez de nouveaux types!
- **typedef**
 - permet de définir des types septiques.
 - permet d'alléger le code et de rendre les programmes plus lisibles

```
typedef struct{
    char sRue [ 100 ] ;
    int iCodePostal ;
    char sVille [ 20 ] ;
} Adresse;

Adresse ad1;
```

Exercice

- Ecrire le programme permettant de définir une structure représentant les informations d'un étudiant : nom, prénom, adresse, diplôme préparé, mention, année dans le cursus.

Exercice

```
1  #include <stdio.h>
2  #define TLABEL 30
3  #define TLABEL_LONG 100
4
5  typedef struct {
6      char sDiplome [TLABEL] ;
7      char sMention [TLABEL] ;
8      short iAnnee ;
9  } Inscription;
10
11 typedef struct{
12     char sRue [TLABEL_LONG] ;
13     int iCodePostal;
14     char sVille [TLABEL] ;
15 } Adresse ;
16
17 typedef struct{
18     char sNom[TLABEL];
19     char sPrenom[TLABEL];
20 }Identite;
```

```
typedef struct{
    Identite id;
    Adresse ad;
    Inscription insc;
}Etudiant;

int main(){

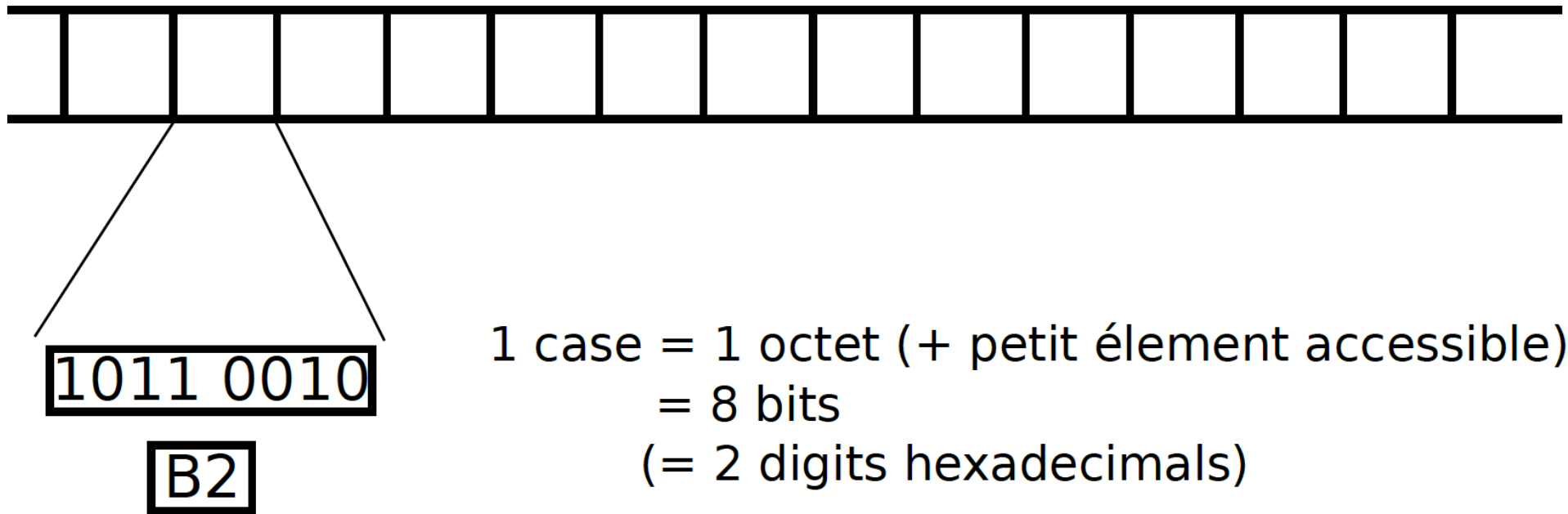
    Etudiant ed1;

    return 0;
}
```

Pointeurs

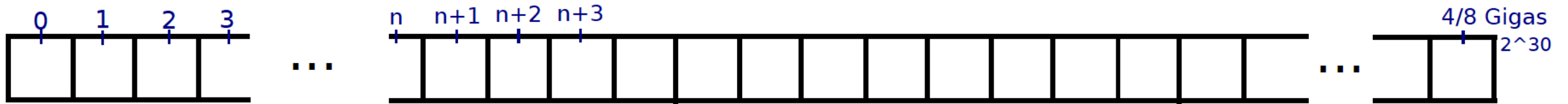
Organisation mémoire

- Mémoire RAM = Suite élément mémoire



Organisation mémoire

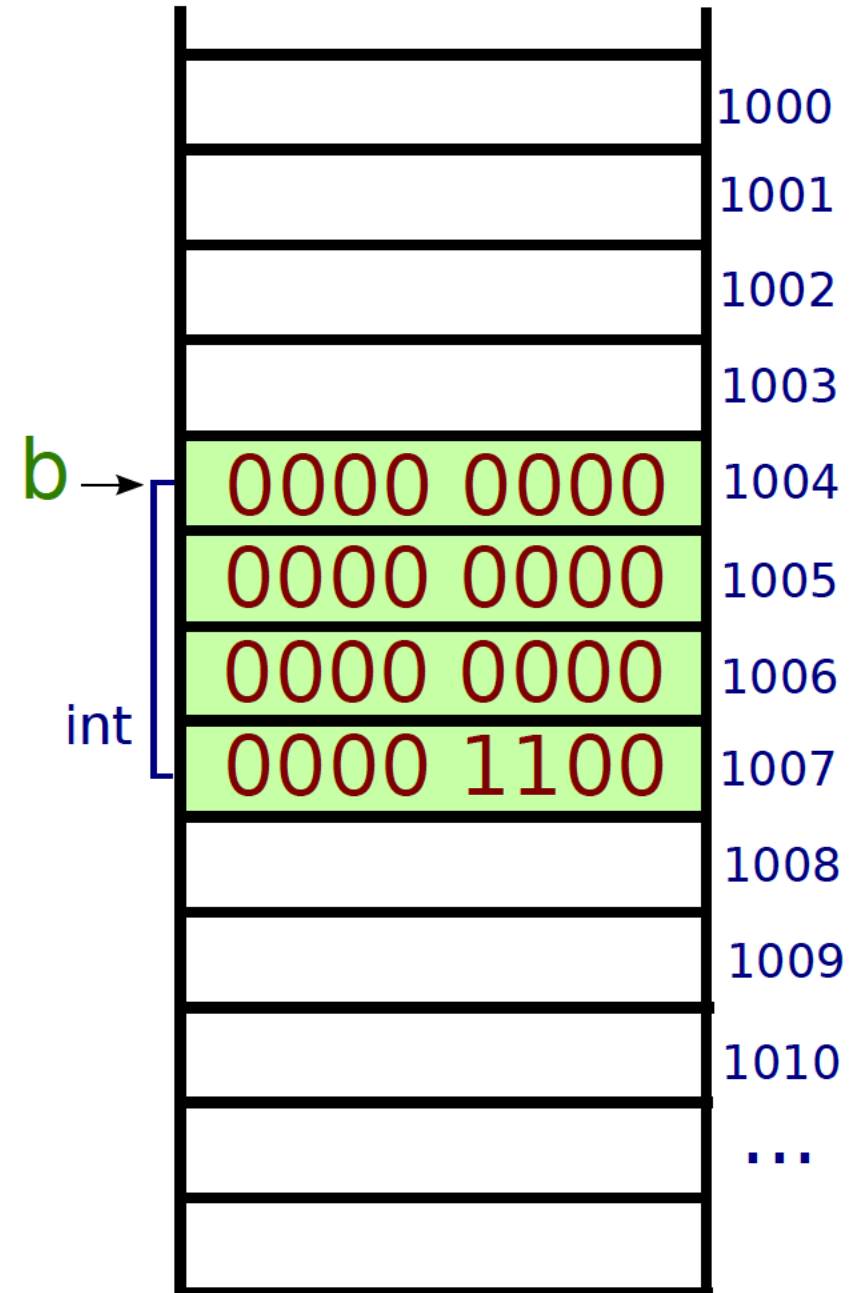
- Chaque case est numérotée
 - C'est son adresse



Rem. Pointeur NULL = Pointeur vers la case numéro 0

Variable

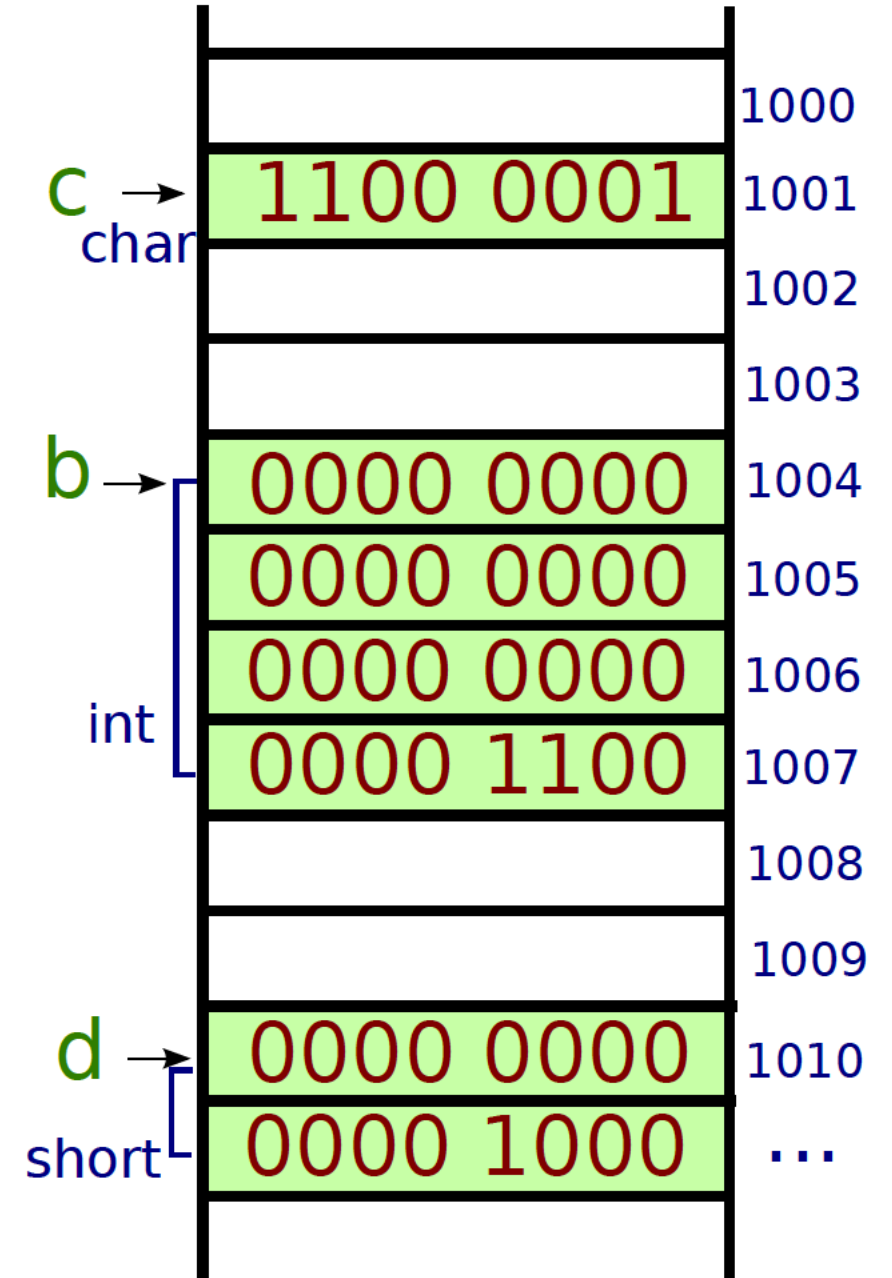
- Instancier une variable revient à:
 - Réserver une case en mémoire
 - La taille de la case est donnée par le type
 - Int -> 4 Octets
 - Désigner cette case par le nom de variable
 - Remplir la case par la valeur désignée
- `int b=12;`



Variable

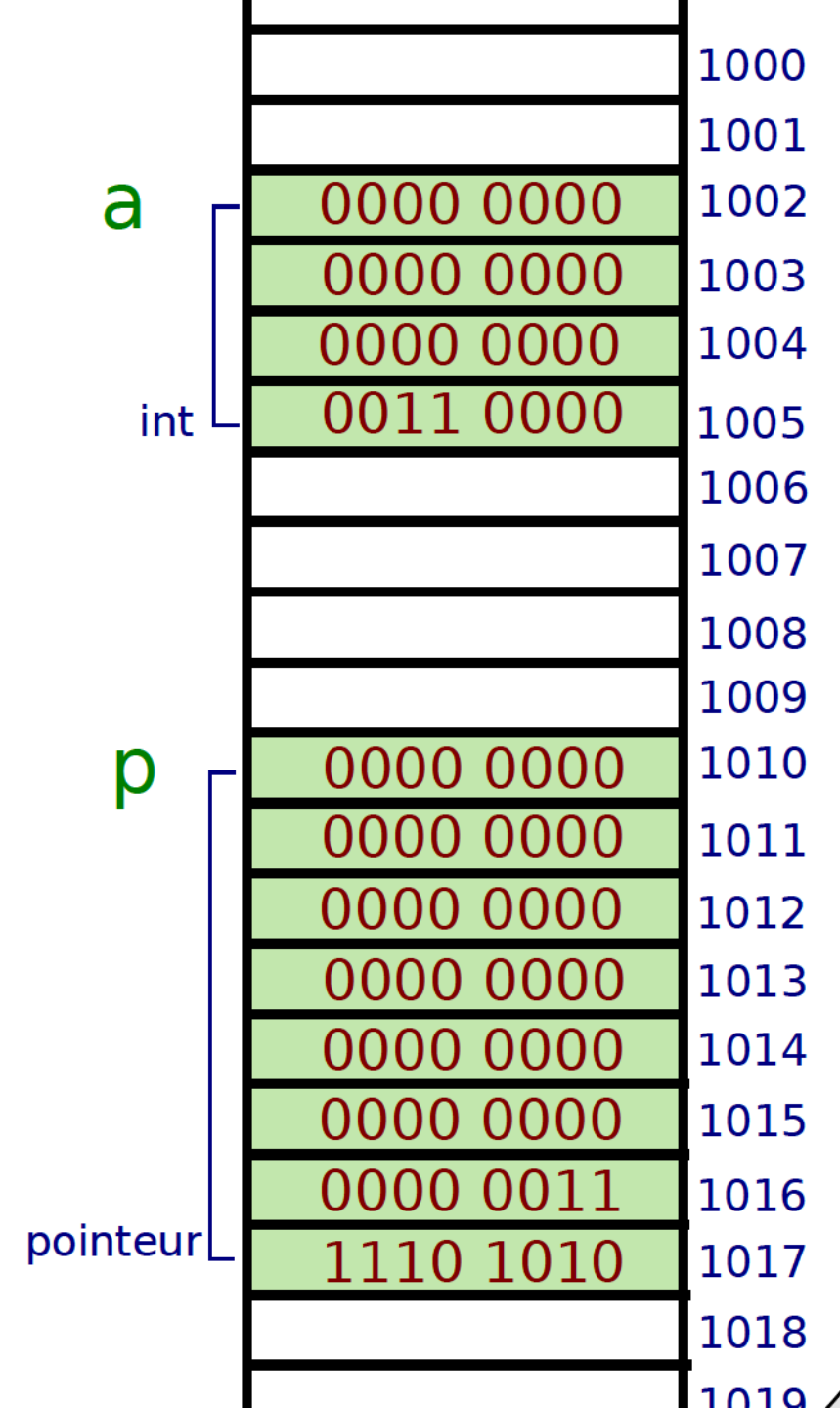
- Lorsqu'il y a plusieurs variables, elles ne sont pas forcément contiguës.

- int b = 12;
- char c = 'a';
- short d = 8



Pointeur

- Une variable de type **pointeur**:
 - Contient le numéro d'une case mémoire
 - Le type encode la taille du type pointé
 - Est stockée comme une variable de 4/8 octets
- `int a = 48;`
- `int* p = &a;`
- Le type pointé
 - n'est pas nécessaire pour le stockage
 - est utile pour lire la valeur de `*p`



Pointeur

- L'opérateur **&** fournit l'adresse de son opérande
 - *int a ;*
 - */ &a est du type pointeur vers un entier */*
 - */ &a est une adresse d'entier */*
- L'opérateur ***** fournit l'objet pointé par son opérande
- */* Si ptr est de type pointeur sur un entier, *ptr est de type entier */*

Pointeur

- Quel que soit un type T , on peut créer un type pointeur vers T
 - La partie réservée à la taille est nécessaire et suffisante pour contenir une adresse.
 - La déclaration d'une variable pointeur fait toujours intervenir le type des objets pointés, on dit que le pointeur est typé.

• $T * \text{ident};$ */* ident est de type pointeur vers T */*

Pointeur : la valeur *NULL*

- **NULL** (toujours en majuscules) est une constante pointeur définie dans `<stdio.h>` et valant *0*.
- Cette valeur signifie « *ne repère aucun objet* ».
- Sorte d'élément neutre
- `int *p = NULL;` */* initialisation propre d'un pointeur */*

Conversion de pointeurs

- Le nom d'une fonction est converti en pointeur sur cette fonction.
- Le nom d'un tableau est converti en pointeur sur son premier élément.
- `int tab [10];`
 - */*tab est un pointeur vers le premier élément du tableau */*

Arithmétique des pointeurs

- Les operateurs classiques de relation sont utilisables avec les pointeurs :
 - `== != < > <= >=`
- Il est possible d'additionner ou soustraire un pointeur avec un entier
 - Le déplacement est alors calculer en fonction du type du pointeur
- `T ptr ;`
- `int i;`
- `/* ptr + i <=> (ptr + i * sizeof(T)) */`

Initialisation

- Comme toute variable en C, un pointeur doit être initialisé pour être utilisé correctement.
- Initialisation **directe** a une adresse :
 - permet d'accéder a des zones spécifiques du système (donc à éviter);
- Initialisation à **une adresse calculée** :
 - *ptr = &i; /*par exemple*/ ;*
- Initialisation par **allocation de mémoire**. Exemple :
 - *int ptr;*
 - *ptr = malloc (100*sizeof(int));* */* défini un tableau de 100 */*

Exercice

- Déclarer un entier i et un pointeur pi ;
- Initialiser l'entier a une valeur arbitraire et faire pointer pi vers i ;
- Imprimer la valeur de i ;
- Modifier l'entier pointé par pi (en utilisant pi , pas i) ;
- Imprimer la valeur de i ;

Exercice

```
#include <stdio.h>

int main(){

    int i = 1;
    int * pi = NULL;

    i = 2;
    pi = &i;

    printf ("La valeur de i avant modification est de : %d", i);

    *pi = 12;|
    printf ("La valeur de i après modification est de : %d", i);

    return 0;
}
```

Les tableaux

Définition

- Un tableau est un **ensemble d'objets** du même type.
- Chaque objet est appelé **élément** du tableau.
- La taille (i.e. le nombre d'éléments) est donnée par une expression entière et positive, et précisée entre crochets [].
 - Cette expression doit être statique, c'est à dire évaluable lors de la phase de compilation.
- *int tab [10];* */* tab est un tableau de 10 entiers */*

Éléments de tableau

- L'implantation en mémoire d'un tableau est contiguë.
- Les éléments du tableau sont indicés de **0** à **(nombre_d_elements – 1)**
 - Attention ! La vérification de non débordement du tableau n'est pas assurée. L'origine ne peut être déplacée comme dans d'autres langages, elle est toujours à 0.

Initialisation

- `int matrice [3] [3] = { 1 , 2 , 0 , 3 } ;`
 - `/ <=> /`
- `int matrice [3] [3] = { {1 ,2 , 0} ,{3 ,0 ,0} , {0 ,0 ,0} } ;`

Omission du nombre d'éléments

- Le **nombre d'éléments** d'un tableau **peut être omis** s'il n'est pas nécessaire au compilateur dans les cas suivants:
 - le tableau est **déclaré mais déjà défini** :
 - *extern int tab[];*
 - **paramètre de fonction** :
 - *void f(int tab[]);*
 - **initialisation** à la définition :
 - *char chaine[] = "coucou";*

Pointeurs et tableaux – éléments communs

- **sizeof(tab)** : taille du tableau (en octet),
- **&tab** : pointeur vers un tableau
 - *&tab == tab* dans la plupart des compilateurs.
- Dans toutes les autres utilisations, *tab* désigne le pointeur vers le premier élément du tableau : ***tab* <=> &tab [0]**

Pointeurs et tableaux – éléments communs

- L'opérateur crochet [], qui permet de désigner un élément d'un tableau, est **automatiquement traduit** (par le compilateur) en un chemin d'accès utilisant le nom du tableau
- on peut donc écrire en C :
 - `"coucou"[3] == 'c' == 3["coucou"] !!`
- **`tab [i] <=> (tab+i)`** */* A ne pas utiliser dans un contexte industriel */*

Attention !

- L'affectation et la comparaison de tableaux n'existent pas en C.
 - Il faut utiliser des fonctions particulières pour pouvoir affecter et comparer des tableaux
- Affectation
 - *tab1 = tab2* /*INTERDIT*/
 - A ne pas faire car cela équivaldrait a faire *&tab1[0] = ...* ce qui est interdit. Il faut utiliser la copie de blocs mémoire ou faire sa propre fonction:
 - ***memcpy*** (*tab1* , *tab2* , *sizeof (tab2)*);

```
void copieTab(int tabSrc[],int tabDest[]){  
    int i;  
    for(i=0; i < TAILLE; i++) {  
        tabDest[i] = tabSrc[i];  
    }  
}
```

Attention !

- Comparaison
 - *tab1 == tab2*
 - Attention ! Compare en fait les 2 adresses de début de tableau *&tab1[0]* et *&tab2[0]* ; ce n'est pas cela que l'on veut.
 - Là aussi il faut utiliser une fonction de comparaison des blocs mémoire :
 - ***memcmp(tab1 , tab2 , max (sizeof (tab1) , sizeof (tab2)))***

Exercice

- Ecrire un programme manipulant un tableau de 5 entiers. Il faudra manipuler un second tableau de la même taille. On pourra saisir le tableau, le copier et l'afficher. Pensez réutilisation!

Exercice

```
#include <stdio.h>

# define TAILLE 5

void saisieTab(int tab[]){
    int i;
    for(i=0; i < TAILLE; i++) {
        printf ("Entrez l element de rang %d du tableau 1\n", i);
        scanf ("%d%c", &tab[i]);
    }
}

void affichTab(int tab[]){
    int i;
    for(i=0; i < TAILLE; i++) {
        printf ("%d ", tab[i]);
    }
    printf("\n");
}

void copieTab(int tabSrc[],int tabDest[]){
    int i;
    for(i=0; i < TAILLE; i++) {
        tabDest[i] = tabSrc[i];
    }
}
```

```
int main(){

    int tab1[TAILLE];
    int tab2[TAILLE];

    saisieTab(tab1);
    affichTab(tab1);
    copieTab(tab1,tab2);
    affichTab(tab2);

    return 0;
}
```

Les chaines de caractères

Important !

- Elles **n'existent pas** en C!!!!
- Les chaînes de caractères ne sont utilisables qu'a travers un **tableau de caractères** dont **le dernier élément est le caractère spécial '\0'** (le caractère nul, dont le code ASCII est 0).
- `char chaine[4] = { ' o', 'u', 'i', '\0' } ;`
 - `/* equivalent à : */`
- `char chaine[4] = "oui" ;`

Attention!

- Pour la saisie et l’affichage, utiliser le format : %s
- Attention, pas de ‘&’ dans le *scanf*!
 - pourquoi au fait?
- *char chaine[4];*
- *scanf ("%s", chaine)*

Attention aux définitions!

- Les conséquences des deux définitions (*char chaine[]* et *char *chaine*) ne sont pas les mêmes.

`char ch1[] = "oui";`

ch1[0]	52000	'o'
ch1[1]	52002	'u'
ch1[2]	52004	'i'
ch1[3]	52006	'\0'
	52008	...

`char *ch2 = "oui";`

ch2	72000	80000

	80000	'o'
	80002	'u'
	80004	'i'
	80006	'\0'

Fonctions de manipulation dédiées

- On les trouve dans la librairie `<string.h>`
 - La fonction `gets ()` lit les caractères en entrée jusqu'au moment où elle rencontre `'\n'`,
 - La fonction `puts ()` imprime la chaîne (jusqu'à `'\0'`).

Fonctions de manipulation dédiées

- On les trouve dans la librairie `<string.h>`
 - La fonction `strcpy()` permet de recopier les chaînes de caractères.
 - Pour comparer le contenu de deux chaînes de caractères, on utilise la fonction `strcmp()`.
- Attention, pas de vérification de non débordement!!!!

```
strcpy(chaine, « coucou »);
```

```
if (strcmp(chaine, chaine2)==0)  
    printf (« les chaines sont egales\n»);
```

La gestion de la mémoire

Organisation d'un programme compilé

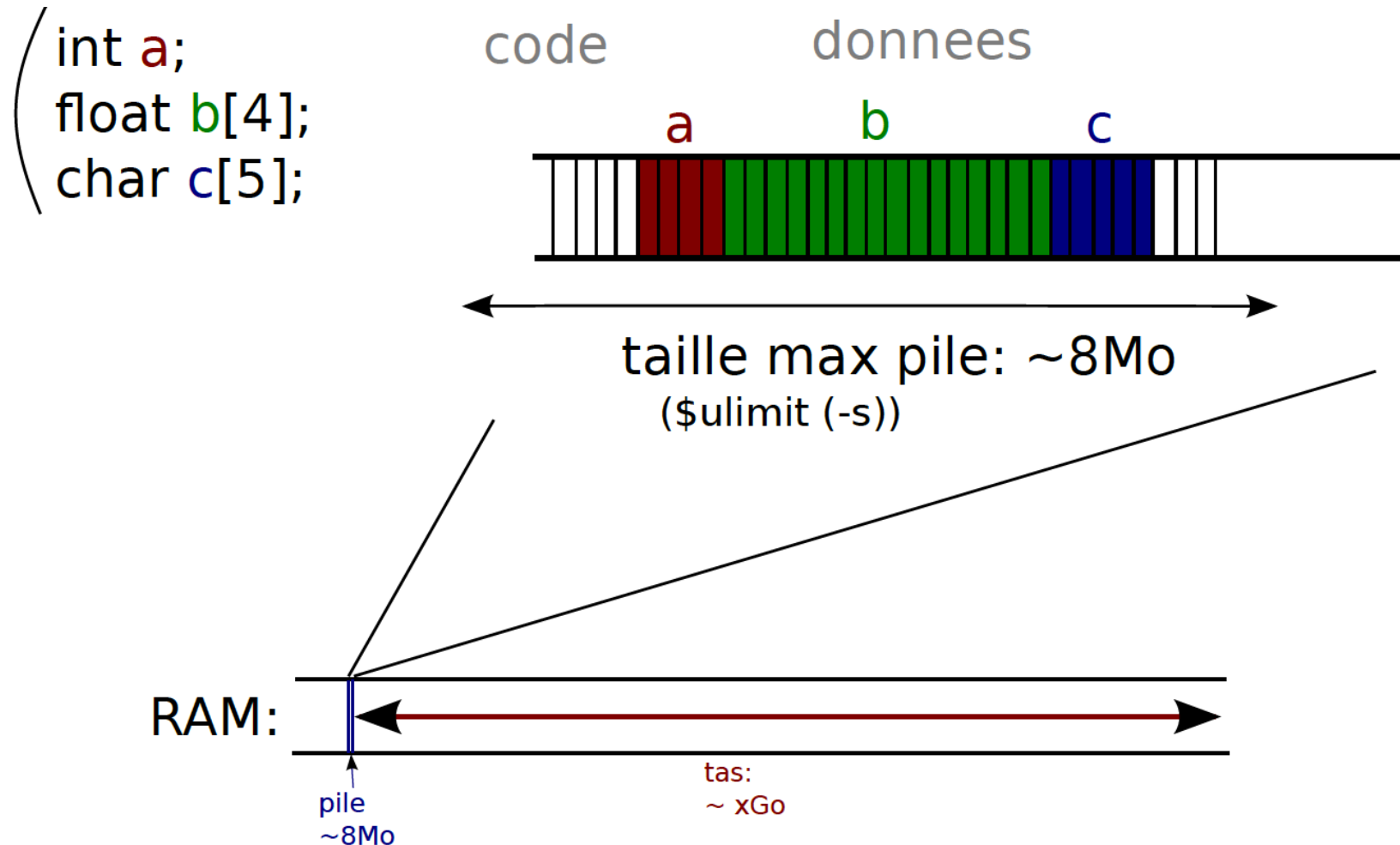
- Un **segment de données** (statique) :
 - emplacement mémoire de tous les objets auxquels le compilateur alloue une **adresse fixe** en mémoire que ce même objet **conservera** durant **toute la durée de vie** du processus.
- Un **segment de code** :
 - emplacement où réside le **code exécutable** fourni par le compilateur.
- La **pile**, le **tas** (dynamique) :
 - emplacement où le processus définit **tous les objets dont la durée de vie est limitée dans le temps** (par opposition au segment de données statique).

Organisation d'un programme compilé

- Attention!!!
 - **Pile** : zone d'allocation des variables automatiques, des paramètres des fonctions, etc.
 - **Tas** : zone d'allocation dynamique

Mémoire dynamique

- Toutes les variables déclarées explicitement sont dans la **pile**



Mémoire dynamique

- Il est possible d'allouer des emplacements mémoire dans le **tas**
 - Avantages:
 - Peut gérer les grandes quantités de données
 - Peut allouer des tableaux de tailles variables
 - Inconvénient:
 - N'est pas géré automatiquement en C
- Allocation/occupation d'espace mémoire par: **malloc(taille)**
- Désallocation/libération d'espace mémoire par: **free(adresse)**
- A la charge du programmeur de **bien gérer** l'allocation/libération
 - => Attention: 90% des erreurs se font sur la gestion mémoire

Mémoire dynamique

```
int main()
{
    int *mon_tableau=NULL;
    mon_tableau=malloc(5*sizeof(int));

    if(mon_tableau==NULL)
        {printf("Erreur allocation memoire\n");exit(1);}

    int k=0;
    for(k=0;k<5;++k)
        mon_tableau[k]=2*k;

    for(k=0;k<5;++k)
        printf("%d\n",mon_tableau[k]);

    free(mon_tableau);
    mon_tableau=NULL;

    return 0;
}
```

Allocation mémoire

Vérification

Utilisation comme un tableau standard

Libération

Mémoire dynamique

```
int main()
{
    int *mon_tableau=NULL;
    mon_tableau=malloc(5*sizeof(int));

    if(mon_tableau==NULL)
        {printf("Erreur allocation memoire\n");exit(1);}

    int k=0;
    for(k=0;k<5;++k)
        mon_tableau[k]=2*k;

    for(k=0;k<5;++k)
        printf("%d\n",mon_tableau[k]);

    free(mon_tableau);
    mon_tableau=NULL;

    return 0;
}
```

Erreur classique : oubli
de la taille des entiers

Erreur classique :
non vérification

Erreur classique :
débordement taille
tableau

Non libération : perte RAM

Mémoire dynamique

Exemple: tableau de taille non connue à la compilation

```
int main()
{
    printf("Donnez un nombre de cases a allouer positif: ");

    int n=0;
    scanf("%d",&n);

    if(n<=0 || n>5000000)
    {
        printf("Nombre %d invalide\n",n);
        exit(1);
    }

    int *tableau=NULL;
    tableau=malloc(n*sizeof(int));

    printf("Je viens d'allouer dynamiquement un tableau de %d entier\n",n);

    int k=0;
    for(k=0;k<n;++k)
        tableau[k]=k;

    free(tableau);
    tableau=NULL;
}
```

Mémoire dynamique

Exemple: redimensionnement d'un tableau

```
int main()
{
    int taille_1=500;
    tableau=malloc(taille_1*sizeof(float));

    if(tableau==NULL)
        {printf("Erreur allocation tableau\n");exit(1);}

    int k=0;
    for(k=0;k<taille_1;++k)//remplissage de 500 cases
        tableau[k]=cos((float)k/taille_1*2*M_PI);

    int taille_2=1000;
    copie_et_agrandissement_tableau(taille_2);

    //remplissage des 500 cases suivantes
    for(k=taille_1;k<taille_2;++k)
        tableau[k]=k*k;

    free(tableau);//liberation de l'espace memoire
    tableau=NULL;

    return 0;
}
```

```
float *tableau=NULL;

void copie_et_agrandissement_tableau(int nouvelle_taille)
{
    float *tableau_temporaire=tableau;//copie du pointeur

    //allocation du tableau avec nouvelle taille
    tableau=NULL;
    tableau=malloc(nouvelle_taille);

    if(tableau==NULL)
        {printf("Erreur allocation tableau\n");exit(1);}

    //copie des valeurs du tableau precedent
    int k=0;
    for(k=0;k<nouvelle_taille;++k)
        tableau[k]=tableau_temporaire[k];

    //liberation du tableau precedent
    free(tableau_temporaire);
    tableau_temporaire=NULL;
}
```

Mémoire dynamique

Exemple: redimensionnement d'un tableau

```
int main()
{
    int taille_1=500;
    tableau=malloc(taille_1*sizeof(float));

    if(tableau==NULL)
        {printf("Erreur allocation tableau\n");exit(1);}

    int k=0;
    for(k=0;k<taille_1;++k)//remplissage de 500 cases
        tableau[k]=cos((float)k/taille_1*2*M_PI);

    int taille_2=1000;
    copie_et_agrandissement_tableau(taille_2);

    //remplissage des 500 cases suivantes
    for(k=taille_1;k<taille_2;++k)
        tableau[k]=k*k;

    free(tableau);//liberation de l'espace memoire
    tableau=NULL;

    return 0;
}
```

```
float *tableau=NULL;
void copie_et_agrandissement_tableau(int nouvelle_taille)
{
    float *tableau_temporaire=tableau;//copie du pointeur

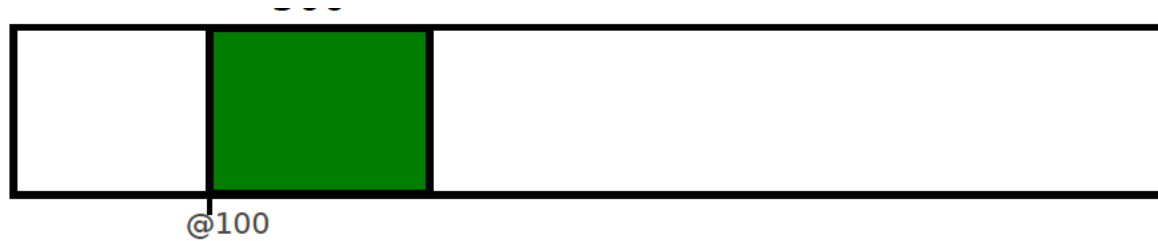
    //allocation du tableau avec nouvelle taille
    tableau=NULL;
    tableau=malloc(nouvelle_taille);

    if(tableau==NULL)
        {printf("Erreur allocation tableau\n");exit(1);}

    //copie des valeurs du tableau precedent
    int k=0;
    for(k=0;k<nouvelle_taille;++k)
        tableau[k]=tableau_temporaire[k];

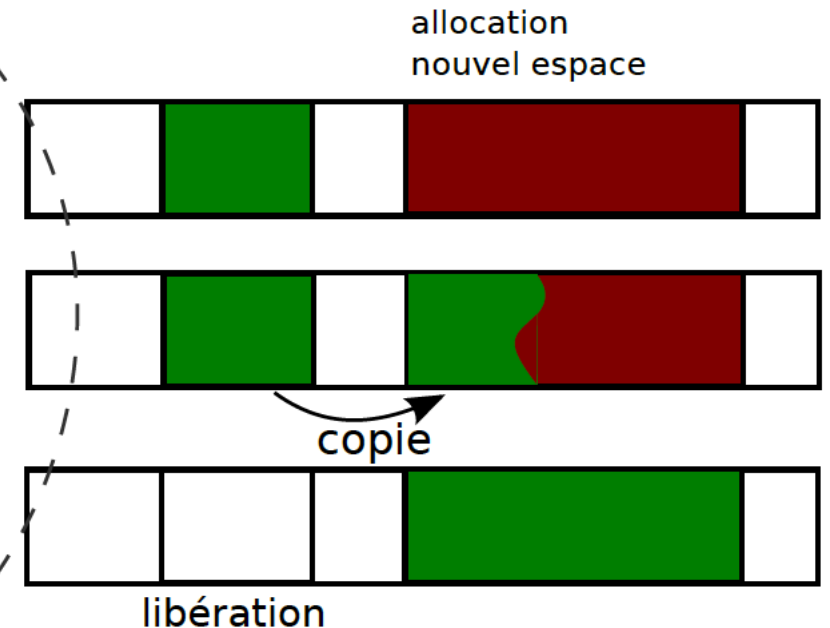
    //liberation du tableau precedent
    free(tableau_temporaire);
    tableau_temporaire=NULL;
}
```

Début

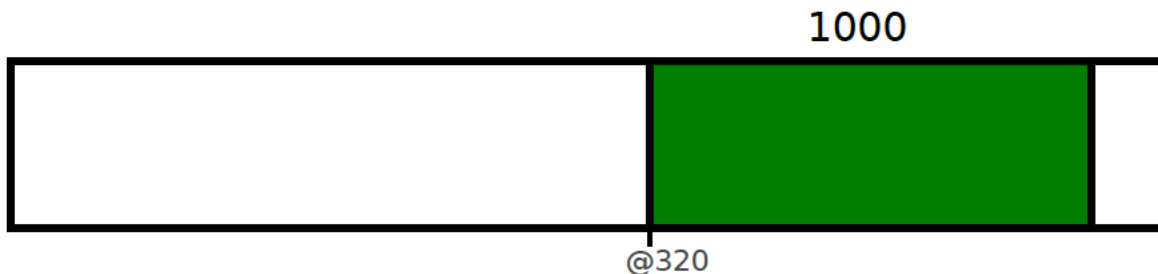


tableau=@100

étapes
intermédiaires



Fin



tableau=@320

Attention: Agrandissement de tableau =
nouvelle allocation + copie = long !

Mémoire dynamique

```
struct tronc
{
    float epaisseur_ecorce;
    int nombre_anneaux;
};

struct feuilles
{
    float largeur;
    float longueur;
};

struct arbre
{
    struct tronc;
    struct feuilles;
};

int main()
{
    struct arbre *foret=NULL;
    foret=malloc(3*sizeof(struct arbre));

    foret[0].tronc.epaisseur_ecorce=1.5;
    foret[1].feuilles.largeur=5.0;

    free(foret);
    foret=NULL;

    return 0;
}
```

Exemple: redimensionnement d'une structure

Les variables

- Une variable possède :
 - Un **nom**,
 - Un **type**,
 - Une **classe d'enregistrement**. Celle-ci définira :
 - le **type d'espace mémoire** (segment de données, pile, ...) dans lequel la variable sera allouée,
 - sa **durée de vie**,
 - sa **zone de visibilité**.

Les variables locales : rappel

- Visibilité
 - Les variables locales ne sont visibles qu'au niveau (fonction, bloc) où elles sont définies.

Les variables locales : classe **auto** (par défaut)

- La variable doit être **allouée** dans la pile **au moment de l'appel à la fonction**.
- L'emplacement mémoire réservé lors de l'allocation **sera libéré lors de la sortie** de l'objet où la variable est définie (fonction ou bloc).

Les variables locales : classe auto (par défaut)

```
char fct1(char a, char b){
    short i;    /* variable locale short int */
    char c;     /* variable locale caractère int */
    /* ... */
    c=fct2();
    return c;   /* valeur retournée */
}
```

```
char fct2(){
    char string [3];
    /*...*/
    return (string[0]);
}
```

*Au retour de fct2(),
l'environnement est
dépilé.
La variable string n'a
plus d'existence.
L'environnement de la
prochaine fonction
appelée écrasera ces va-
leurs*

...
Adresse de retour de fct1()
Code retour de fct1()
char a;
char b;
short i;
char c;
Adresse de retour de fct2()
Code retour de fct2()
string[0];
string[1];
string[2];
...

sauvegarde de l'adresse où re-
prendre le déroulement, à la fin
de cette fonction

paramètres reçus
par fct1()
variables locales
à fct1()
adresse de l'instruction suivant
l'appel à fct2() dans fct1()

variables locales
à fct2()

Les variables locales : classe **static**

- Appelées variables rémanentes.
 - Ces variables sont **allouées dans le segment de données**.
 - Elles **conservent** donc leur **valeur entre deux appels de la fonction** dans laquelle elles sont définies.
 - Leur **domaine de visibilité** reste malgré tout **local** à l'objet où elles sont définies.
 - L'avantage d'un tel type de variable est de pouvoir **initialiser la variable lors de sa déclaration**.

Les variables locales : classe **static**

```
void fct(){
    static i = 5;           /* visibilité limitée à fct */
    printf("%d ",i);        /* affiche 5,6,7,8,9,10 */
}

int main(){
    int i;
    for(i=0;i<7;i++)
        fct();

    return 0;
}
```

Les variables globales

- A l'intérieur d'un programme, il suffit de **déclarer cette variable de niveau fichier**, c'est à dire, en dehors de toute fonction.
- Si l'on fait **référence** à une variable déclarée à l'extérieur d'un programme, il faut utiliser la classe **extern**.

Les fonctions

Quelques points

- En programmation structurée, le programmeur **décompose** le travail à faire en **modules logiques** (ou « sous-programmes »).
- Le concept de sous-programme est directement lié à l'approche de **décomposition fonctionnelle**.
- Un **sous-programme dans le langage C est appelé une fonction** (contrairement à d'autres langages où on différencie procédures et fonctions).
- Le **programme principal** de l'application n'est autre qu'une **fonction** qui doit porter le nom « **main** ».
- Toutes les fonctions d'un programme sont définies au **1^{er} niveau** : elles ne peuvent pas être emboîtées comme en PASCAL par exemple.

Quelques points

- **Tous les éléments, en C, doivent être déclarés avant leur utilisation.** C'est également le cas des fonctions.
- La **déclaration**, ou prototype, permet au compilateur de **vérifier** que la valeur retournée est bien du même type à l'utilisation et à la définition.

```
float fct();           /*declaration de la fonction fct */

int main(){
    float a;
    a = fct();         /* Utilisation de la fonction fct */

    return 0;
}

float fct(){           /* corps de la fonction fct */
    /* ... */
}
```

Éléments grammaticaux

- Les **actions** réalisées par une fonction sont **regroupées dans un bloc**.
- Le **retour** est effectuée par :
 - la rencontre de **la fin du corps** de la fonction
 - ou par la rencontre d'une instruction **return**.

```
int plus(int a, int b){
    int c;
    c = a + b;
    return c;
}

void afficheInt(int a){
    printf("%d\n", a);
}

int main(){
    int i = 2, j = 3;
    int sum = plus (a,b);
    afficheInt(sum);
    return 0;
}
```

Important!!!!

- En C, il n'existe que le **passage par valeur**.
- C'est à dire que la valeur des **paramètres effectifs** (lors de l'appel) **sont copiés** dans un emplacement local a la fonction, qui travaille avec cette copie.
- Il n'y a **pas copie inverse** a la sortie de la fonction (en dehors du retour de la fonction).

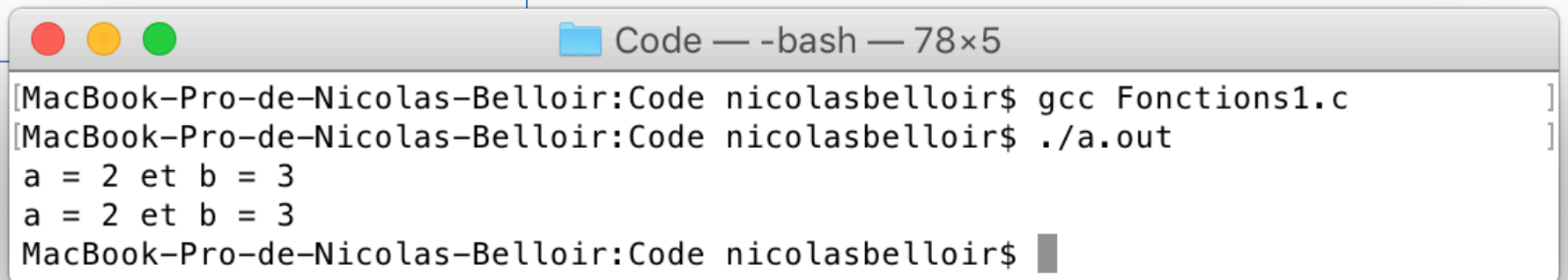
Exemple

```
void echange (int i, int j){
    int tmp;
    tmp = i;
    i = j;
    j = tmp;
}

int main(){
    int a = 2, b = 3;


    printf("a = %d et b = %d\n", a, b);
    echange (a, b);
    printf("a = %d et b = %d\n", a, b);
    return 0;
}
```

- Ecrire une fonction qui échange 2 valeurs entières




```
Code — -bash — 78x5
[MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$ gcc Fonctions1.c
[MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$ ./a.out
a = 2 et b = 3
a = 2 et b = 3
MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$
```

Voyons coté mémoire



```
void echange (int i, int j){  
    int tmp;  
    tmp = i;  
    i = j;  
    j = tmp;  
}  
  
int main(){  
    int a = 2, b = 3;  
  
    printf("a = %d et b = %d\n", a, b);  
    echange (a, b);  
    printf("a = %d et b = %d\n", a, b);  
    return 0;  
}
```



tmp	2
j	2
i	3
Adresse retour echange	
b	3
a	2
Adresse retour main	

Solution!!!!

- Lorsque l'on souhaite qu'une fonction **modifie des paramètres effectifs**, **on ne va pas transmettre leur valeur mais leur **adresse****.
- Ainsi la fonction pourra accéder à la zone mémoire à modifier.

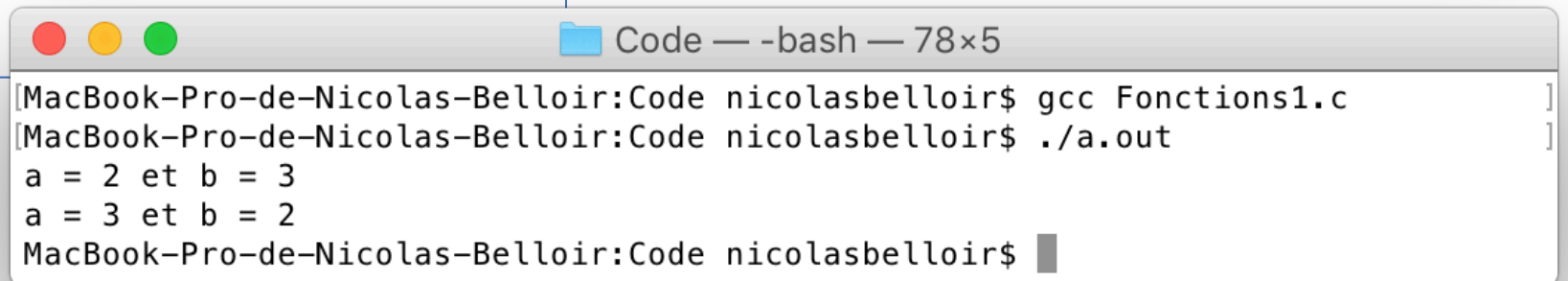
Exemple

```
void echange (int * pi, int * pj){
    int tmp;
    tmp = * pi;
    * pi = * pj;
    *pj = tmp;
}

int main(){
    int a = 2, b = 3;

    printf("a = %d et b = %d\n", a, b);
    echange (&a, &b);
    printf("a = %d et b = %d\n", a, b);
    return 0;
}
```

- Ecrire une fonction qui échange 2 valeurs entières



```
Code — -bash — 78x5
[MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$ gcc Fonctions1.c
[MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$ ./a.out
a = 2 et b = 3
a = 3 et b = 2
MacBook-Pro-de-Nicolas-Belloir:Code nicolasbelloir$
```


Voyons coté mémoire

```
void echange (int * pi, int * pj){
    int tmp;
    tmp = * pi;
    * pi = * pj;
    *pj = tmp;
}

int main(){
    int a = 2, b = 3;

    printf("a = %d et b = %d\n", a, b);
    echange (&a, &b);
    printf("a = %d et b = %d\n", a, b);
    return 0;
}
```

tmp	2	
pj	52404	
pi	52400	
Adresse retour echange		
b	2	52404
a	3	52400
Adresse retour main		

Remarque

- Cette approche est également utile lorsque les **paramètres** à transmettre sont **volumineux** (paramètres structures ou tableaux).
- Pour **transmettre un tableau** en paramètre, il suffit de **transmettre l'adresse de son premier élément** (et parfois la **taille** du tableau si nécessaire) pour que la fonction y accède.

```
int trier (int * tab, int iTaille){  
    /*...*/  
}
```

Allocation dynamique et paramètres

- Comment transmettre une variable allouée dans une fonction?
- Pour **modifier dans une fonction** une **variable de type entier**, on passe à la fonction un **paramètre de type pointeur** sur entier.

```
int i;          void fct ( int * pi ) { /*...*/ }
```

- Pour **modifier dans une fonction** une **variable de type pointeur sur entier**, on passe à la fonction un **paramètre de type pointeur de pointeur** sur entier.

```
int * i;          void fct ( int * * pi ) { /*...*/ }
```

Voici le temps du pointeur de pointeur!!!!!!

Allocation dynamique et paramètres

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

int creerPointeur (int ** pi){
    *pi = malloc(sizeof (int));
    if(*pi==NULL)
        return -1;
    return 0;
}
```

```
int main(){
    int * pi = NULL;
    int iRetour = 0;

    iRetour = creerPointeur(&pi);
    if (iRetour){
        printf("ERREUR - fin du program\n");
        return(-1);
    }
    printf("Mémoire allouée\n");
    free(pi);

    return 0;
}
```