

PROGRAMMATION AVANCÉE

PHUC NGO

CONTENUE DU COURS

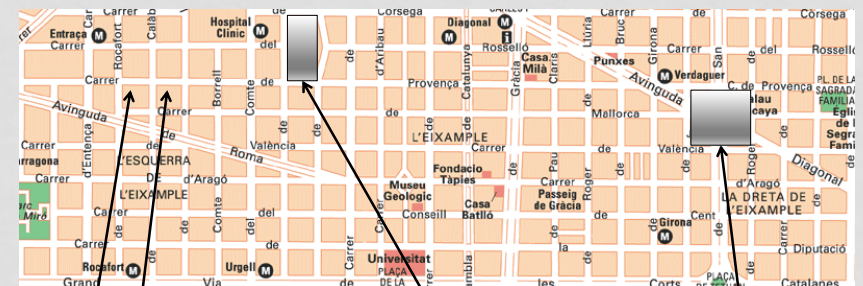
- Rappels
 - Fonctions et procédures
 - Structures de données
 - Enregistrements, tableaux, liste enchainé, Pile (FIFO), file (FILO), ...
- Compilation avec Makefile/Cmake et la bibliothèque SDL
- **Gestion de mémoire**
 - Représentation de mémoire
 - Pointeurs et allocation dynamique
- Gestion de fichiers
 - Lecture et écriture
- Gestion des entrées et sorties
 - Ecran, souris et clavier
- Directives au préprocesseur
- Structures de données et algorithmes de graphes
 - Parcours de graphe, arbre couvrant, ...

REPRÉSENTATION DE MÉMOIRE



Barcelone
<http://carreephemere.blogspot.com>

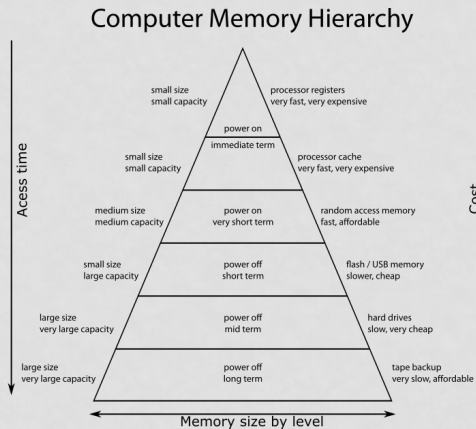
REPRÉSENTATION DE MÉMOIRE



78 Carrer de Provença
80 Carrer de Provença
Hospital Clínic de Barcelona
170-172 Carrer de Villarroel
Place Mossèn Jacint Verdaguer
Barcelone
<http://carreephemere.blogspot.com>

HIÉRARCHIE MÉMOIRE

- Différentes type de mémoires sur ordinateur
 - Registres de processeur
 - Mémoire cache CPU
 - Mémoire vive (DRAM)
 - Disque dur
- Les temps d'accès sont inversement proportionnels au coût
- Moins une mémoire est grande, plus elle est rapide et plus elle est chère



wikipédia

HIÉRARCHIE MÉMOIRE

Type de mémoire	Utilité	Taille
Registres de processeur (mémoire interne)	Stocker l'état du processeur, le jeu d'instructions, ...	qq milliers d'octets
Mémoire cache CPU (SRAM)	Enregistrer temporairement des copies de données afin d'accélérer le temps d'accès	6 Ko – 128 Mo
Mémoire vive (DRAM)	Enregistrer les données / instructions traitées des programmes informatiques	qq Go
Disque dur	Stocker de données de longues durées sur ordinateur	qq To

Autres : USB, disque dur extern, CD-ROM, DVD, cloud, ...

→ La mémoire permet au processeur d'accéder aux instructions du programme à exécuter et aux données nécessaires à son exécution

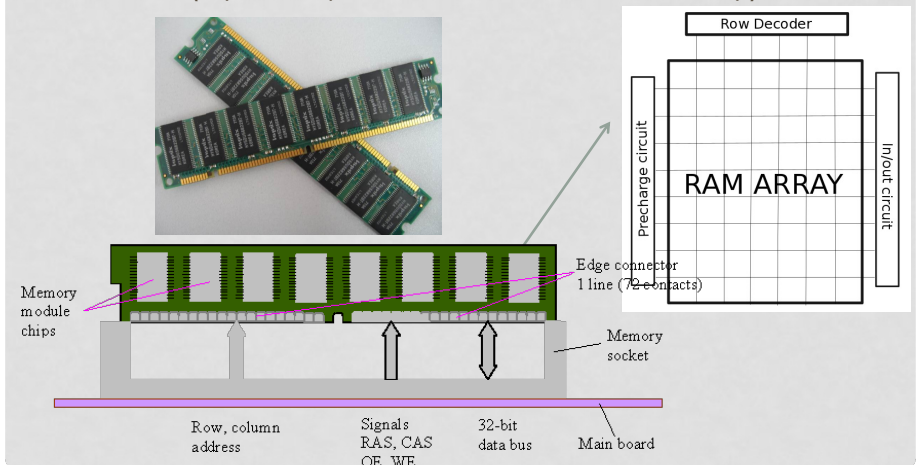
TEMPS D'ACCÈS À LA MÉMOIRE

Type de mémoire	Taille	Temps d'accès
Registres de processeur	qq milliers d'octets	1 cycle processeur (~ ns)
Mémoire cache CPU (SRAM)	N0 : mémoire des microcodes	6 Ko
	N1 : mémoire cache des instructions / données	128 Ko
	N2,3,4 : cache partagée des instructions / données	1, 6, 128 Mo
Mémoire vive (DRAM)	qq Go	10 Go/seconde
Disque dur	qq To	~ 75 Mo/seconde

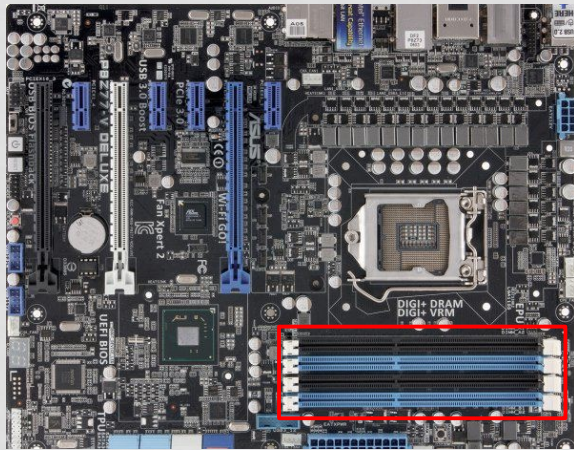
- L'accès au disque dur est **beaucoup fois plus lente** qu'à la mémoire vive
- L'exécution des programmes informatiques se fait avec le mémoire vive

MÉMOIRE SUR ORDINATEUR

- DRAM (dynamique random access memory)



MÉMOIRE SUR ORDINATEUR



Carte mère
<https://www.conseil-config.com>

MÉMOIRE SUR ORDINATEUR

- La mémoire dans un ordinateur a une taille finie
 - 1 Go, 2 Go, 3 Go, 4 Go, ...
- La mémoire est limitée en fonction de
 - Nombre de slot sur la carte mère
 - 2 slots ou 4 slots
 - Système de gestion de la mémoire (Memory manager)
 - Système de 32 bits (2^{32} adresses) \approx 4 Go de RAM
 - Système de 64 bits (2^{64} adresses) \approx 16 Eo de RAM
 - Système d'exploitation

Édition de Windows 10	Mémoire RAM maximale
Windows 10 Home 32 bit	4 GB
Windows 10 Home 64 bit	128 GB
Windows 10 Pro 32 bit	4 GB
Windows 10 Pro 64 bit	512 GB

Wikipédia

ADRESSE ET VALEUR

- Mémoire est organisé en suite d'octets = cases
- Chaque case a numéro en hexadécimal = **adresse**
- Chaque case contient une valeur = **contenu**
 - 01100001 ⇔ 'a' ou 97
 - 10101010 ⇔ 170 ou -86
- Exemple: char x='a'
 - Adresse de x est 6012
 - Valeur de x est 01100001 ⇔ 'a'

	Adresse	Valeur
0		
1	0000	1203
	0002	FFFE

	6012	'a'


	FFFE	9
F	FFFF	4210

TAILLE DES TYPES

- Une variable possède une seule adresse et une adresse correspond à une seule variable
 - Chaque type de données s'occupe un certain nombre de cases
 - char = 1 octet = 1 case
 - int = 4 octets = 4(*) cases
 - float = 4 octets = 4(*) cases
 - L'adresse de la 1^{ère} case
- (*) système 32 ou 64 bits

0000	0001	0FFF
1000	1001	1FFF
F000	F001	FFFF

TAILLE DES TYPES

- Pour savoir la taille d'un type sur le système
 - Fonction : **size_t sizeof** (type)
 - Ex : printf(" taille d'un entier en octets est %d\n", sizeof(int));
- Conversion de type : (type) variable 

Système 32 ou 64 bits

Type de Base	Taille en octets	Nombres de valeurs possibles
unsigned char	1	256 : [0, 255]
(signed) char	1	256 : [-128, 127]
unsigned (int)	4	4 294 967 296
(signed) int	4	4 294 967 296
float	4	4 294 967 296
double	8	18 446 744 073 509 551 616
long double	12	79 008 162 513 705 374 134 343 950 336

TAILLE DES TYPES

Système 32 bits

Type de Base	Taille en octets	Nombres de valeurs possibles
unsigned long (int)	4	4 294 967 296
signed long (int)	4	4 294 967 296
unsigned long long (int)	8	18 446 744 073 509 551 616
(signed) long long (int)	8	18 446 744 073 509 551 616

Système 64 bits

Type de Base	Taille en octets	Nombres de valeurs possibles
unsigned long (int)	8	18 446 744 073 509 551 616
(signed) long (int)	8	18 446 744 073 509 551 616

OPÉRATEURS

- Les valeurs entières
 - Opérateurs arithmétiques (+, -, *, / et %)
 - Opérateurs relationnels (>, >=, <, <=, == et !=)
 - Opérateurs d'affectation composée (+=, -=, *=, /=, %=)
 - Opérateurs d'incrément et de décrémentation (v++, ++v, v--, --v)
- Les valeurs réelles
 - Opérateurs arithmétiques (+, -, *, / et PAS DE %)
 - Opérateurs relationnels (>, >=, <, <=, == et !=)
 - Opérateurs d'affectation composée (+=, -=, *=, /=)
 - Opérateurs d'incrément et de décrémentation (v++, ++v, v--, --v)

OPÉRATEURS

- Les valeurs booléennes
 - **#include <stdbool.h>**
 - Ou logique (||), et logique (&&), non logique (!)
- Opérateurs bit-à-bit
 - Ou (|), et (&), ou exclusif (^)
 - Exemple : 9 & 12 = 8 (1001 & 1100 = 1000)
- Opérateurs de décalage de bit
 - Décaler à droite (>>), et à gauche (<<)
 - Exemple : 6 >> 1 = 3 (0110 >> 1 = 0011)
- Opérateur conditionnel ternaire (condition ? v1 : v2)
 - Exemple : bool pair = x % 2 == 0 ? true : false
- Opérateur adresse (&variable)
- Opérateur unaire indirection (*pointeur)

CONSTANTES

- Définir les constantes en C avec
 - La directive du préprocesseur : **#define** Const Valeur
 - Le mot-clé : **const** type variable = valeur ;
- Constantes entières : 3 manières différentes
 - Décimale : base 10. Ex : 14 et 255
 - Octale : base 8. Ex : **016** et **0377**
 - Hexadécimale: base 16, contient 0-9a-f. Ex : **0xe** et **0xff**
- Indication explicite le type d'une constante entière

Constantes	Types
1234	int
02322	int /* octal */
0x402	int /* hexadécimal */
123456789L	long
1234U	unsigned int
123456789UL	unsigned long int

CONSTANTES

- Constantes réelles sont représentées par mantisse et exposant
 - L'exposant est écrit par e ou E. Ex: 12.3e-2 = 0.123
- Indication explicite le type d'une constante réel

Constantes	Types
12.34 ou 0.1234e2 ou 1234e-2	double
12.34F	float
12.34L	long double

- Définition des constantes en C
 - Directive du préprocesseur : **#define** PI 3.14
 - Mot clé : **const** float PI = 3.14;

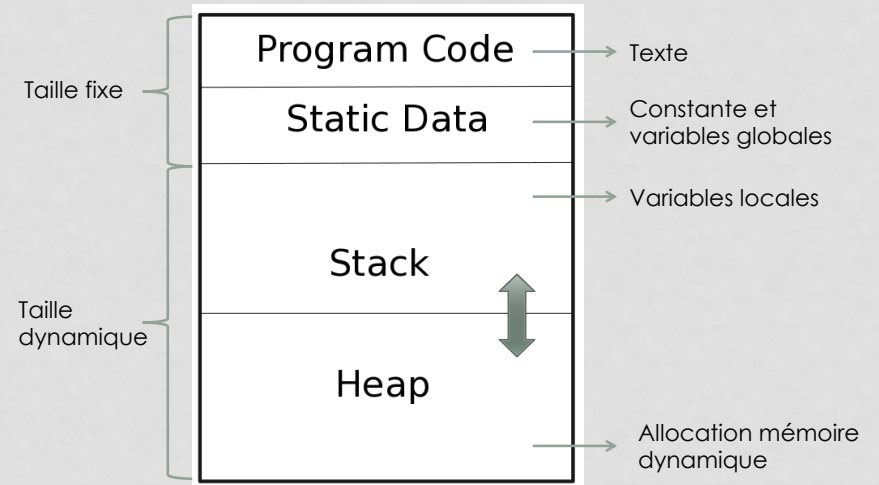
CONSTANTES

- Constante caractère est entouré entre apostrophes
 - Caractères imprimables. Exemple : a, B, 1, 2
 - Caractères spéciales. Exemple : `\\`, `\'`, `\?`, `\"`
 - Caractères non-imprimables

Caractères	Significations
<code>\n</code>	nouvelle ligne
<code>\r</code>	retour de la ligne
<code>\t</code>	tabulation horizontale
<code>\f</code>	saut de page
<code>\v</code>	tabulation verticale
<code>\a</code>	signal d'alerte
<code>\b</code>	retour arrière

- Constante chaîne de caractères est entouré entre guillemet. Ex : "Cours `\t` C `\n` "

PROGRAMME

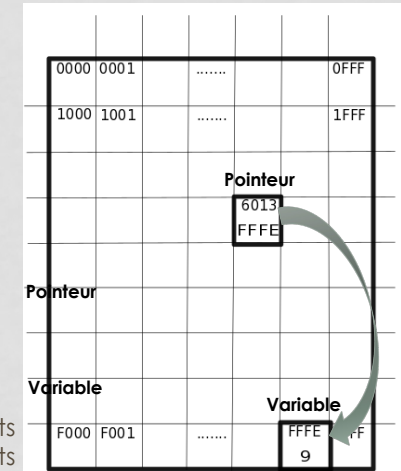


PROGRAMME

- La pile (stack)
 - Stockage des paramètres d'appel et des variables locales des fonctions
 - Un espace mémoire réservé au stockage des variables qui sont désallouées automatiquement
- Le tas (heap)
 - Allocation dynamique de mémoire durant l'exécution d'un programme
 - La mémoire allouée dans le tas doit être désallouée explicitement
- Visualiser facilement les valeurs du tas et de la pile
 user:~\$ ulimit -a
 ...
 stack size (kbytes, -s) 8192 → "Erreur de segmentation"
 ...

POINTEUR ET ADRESSE

- Stocker une adresse
 - Une variable qui contient l'adresse d'une autre
 - Tableau = pointeur !!!
- Partager une variable
 - Une variable = une adresse
 - Plusieurs pointeurs peuvent pointer sur la même variable
- Passer la variable en fonction
 - Passage de paramètres par référence
 - Modification de la valeur de la variable après la fonction
- Pointeur = size_t
 - 4 octets sur un système de 32 bits
 - 8 octets sur un système de 64 bits



POINTEUR SUR OBJETS

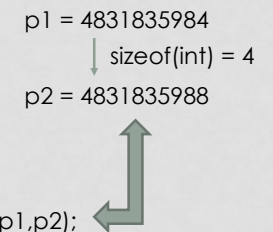
- Déclarer un pointeur = comme une variable(*)
 - Type du pointeur
 - Nom du pointeur
- Syntaxe en C/C++
 type_données* nom_pointeur
- Initialiser un pointeur avec l'adresse de la variable
 nom_pointeur = &nom_variable
- Récupérer la valeur de la variable pointée
 *nom_pointeur == nom_variable
- Exemple


```
int x = 3;
int *y = &x;  ← y = @x
*y = 5        ← x = 5
```

POINTEUR SUR OBJETS

- Opérateurs arithmétiques des pointeurs
 - Addition d'un entier à un pointeur
 - Soustraction d'un entier à un pointeur
 - Incrémentation (++) et décrémentation (--)
 - Différence de deux pointeurs pointant vers des objets de même type
- Attention :
 - Type *p; int i;
 - $p + i \Leftrightarrow p + i * \text{sizeof}(\text{Type})$
 - Exemple :


```
int i = 3;
int *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n", p1, p2);
```



POINTEUR SUR OBJETS

- Opérateurs arithmétiques des pointeurs
 - Addition d'un entier à un pointeur
 - Soustraction d'un entier à un pointeur
 - Incrémentation (++) et décrémentation (--)
 - Différence de deux pointeurs pointant vers des objets de même type

- Attention :

- Type *p; int i;
- $p + i \Leftrightarrow p + i * \text{sizeof}(\text{Type})$
- Exemple :

```
double i = 3;
double *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n", p1, p2);
```

```
p1 = 4831835984
      ↓ sizeof(double) = 8
p2 = 4831835992
```

POINTEUR SUR OBJETS

- Opérateurs arithmétiques des pointeurs
 - Addition d'un entier à un pointeur
 - Soustraction d'un entier à un pointeur
 - Incrémentation (++) et décrémentation (--)
 - Différence de deux pointeurs pointant vers des objets de même type

- Attention :

- Type *p; int i;
- $p + i \Leftrightarrow p + i * \text{sizeof}(\text{Type})$
- Type *p1, *p2 ;
- $p1 - p2 \Leftrightarrow (p1 - p2) / \text{sizeof}(\text{Type})$

EXEMPLES : POINTEURS

```
int x = 3;
int y;
int* z;
y = x;
z = &x; → z = @x
printf("x=%d, y=%d et *z=%d\n", x, y, *z); → x=3, y=3 et *z=3
```

```
y = x++;
x += 3;
printf("x= %d, y=%d et *z=%d\n", x, y, *z); → x=7, y=3 et *z=7
```

```
x = ++y;
*z = 20; ↔ x = 20
printf("x= %d, y=%d et *z=%d\n", x, y, *z); → x=20, y=4 et *z=20
```

```
int* w = z; ↔ w = z = @x
*w = 2; ↔ (*w) = (*w)*2
printf("x= %d, y=%d, *z=%d et *w=%d\n", x, y, *z, *w); → x=40, y=4, *z=40 et *w=40
```

QUIZZ : POINTEURS

```
double a = 5;
double *b = &a; → b = @a;
double c = *(&a); → c = a = 5;
```

```
int *p1, p2; → Un pointeur int (p1) et une variable int (p2)
```

```
int *p3 = 0;
int p4 = *p3;
*p3 = 12; → Segmentation fault
```

```
int n = 1, p = 2;
int *ad1, *ad2;
ad1 = &n; → ad1 = @n
ad2 = &p; → ad2 = @p
*ad1 = *ad2 + 3; → n = p + 3
ad1 = ad2; → ad1 = ad2 = @p
*ad1 = *ad2 + 5; → p = p + 5
printf("n=%d et p=%d\n", n, p); → n=5 et p=7
```


UTILISATION DE POINTEUR

- Pointeur sur les tableaux
 - `int tab[5];`
 - `int* pt = tab;`
 - `tab[0] ⇔ *tab ⇔ *pt`
 - `tab[5] ⇔ *(tab+5) ⇔ *(pt+5) != *tab+5 ⇔ *pt+5`
 - Et `int** p1, float*** p2, ... ?!`
- Les enregistrements
 - `personne p, personne* pp=&p;`
- Envoyer un pointeur à une fonction
 - Paramètres → `type*`
 - Appel de la fonction → `&variable` SAUF pour les tableaux !
- Paramètre par valeur et par référence

EXEMPLES : TABLEAUX

```
int T[10];
for(int i = 0; i < 10; i++)
    T[i] = i * i;
for(int i = 0; i < 10; i++)
    printf( "%d ", T[i]);
    → 0 1 4 9 16 25 36 49 64 81

int * p = T, * q = &(T[2]);
printf( "**p=%d, p[0]=%d et p[1]=%d\n", *p, p[0], p[1]);
    → *p=0 , p[0]=0 et p[1]=1
p++;
printf( "**p=%d, p[0]=%d et p[1]=%d\n", *p, p[0], p[1]);
    → *p=1 , p[0]=1 et p[1]=4
printf( "**q=%d, q[0]=%d et q[1]=%d\n", *q, q[0], q[1]);
    → *q=4 , q[0]=4 et q[1]=9
p = q + (q - p);
printf( "**p=%d, p[0]=%d et p[1]=%d\n", *p, p[0], p[1]);
    → *p=9 , p[0]=9 et p[1]=16
```

VISIBILITÉS DES VARIABLES

Variables	Visibilité
Variables globales	Tout le programme(*)
Variables locales	Localement
• D'un bloc de code {...}	• Dans le bloc
• D'une fonction	• Dans la fonction
Paramètre d'une fonction	Modification affectée
• Passage par valeur (variable)	• Dans la fonction
• Passage par référence (pointeur)	• Dans et en dehors de la fonction

EXEMPLES : VISIBILITÉS DES VARIABLES

```
int x, y; → x et y sont les variables globales
int somme ( int x, int y ) { → x et y sont les paramètres
    return x + y;
}
int moyenne ( int a, int b ) {
    int x = somme(a, b); → x et y sont les variables locales
    int y = x / 2;
    return y;
}
void main( void ) {
    x = 5, y = 7; → x et y sont les variables globales
    printf("x=%d et y=%d", x, y); → x=5 et y=7
    for (int i=0; i<3; i++) {
        int x = somme (i,i); → x et y sont les variables locales
        int y = moyenne (i,i);
        printf("x=%d et y=%d", x, y); → x=0 et y=0
    } → x=2 et y=1
    printf("x=%d et y=%d", x, y); → x=5 et y=7 → x=4 et y=2
}
```


EXEMPLES : PASSAGE DES PARAMÈTRES

```
void multiple2_valeur( int x ) {
    x*=2;  ↔  x=x*2
}
// Passage de paramètre par valeur

void multiple2_pointeur( int* x ) {
    *x*=2;  ↔  *x=(*x)*2
}
// Passage de paramètre par référence

void main( void ) {
    int n=3;
    multiple2_valeur(n);
    printf( "Appel multiple2_valeur : %d\n", n ); → n=3
    multiple2_pointeur(&n);
    printf( "Appel multiple2_pointeur : %d\n", n ); → n=6
}
```

EXEMPLES : PASSAGE DES PARAMÈTRES

```
int multiple2_valeur( int x ) {
    return 2*x;
}
// Passage de paramètre par valeur

int multiple2_pointeur( int* x ) {
    *x*=2;
    return *x;
}
// Passage de paramètre par référence

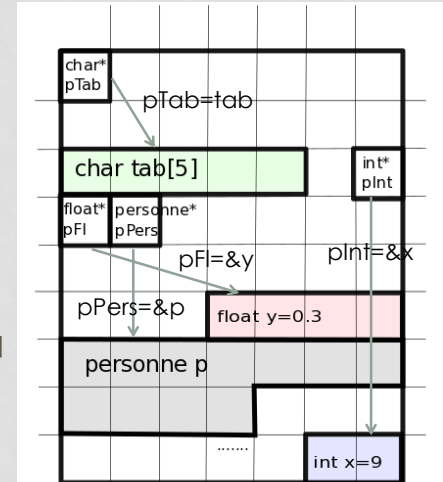
void main( void ) {
    int n=3;
    int m= multiple2_valeur(n);
    printf( "Appel multiple2_valeur : n=%d et m=%d", n, m ); → n=3 et m=6
    int p= multiple2_pointeur(&n);
    printf( "Appel multiple2_pointeur : n=%d et p=%d", n, p ); → n=6 et p=6
}
```

EXEMPLES : POINTEUR

```
char ch[] = "Bonjour";
char *pc = ch;
*pc='N';
pc++;
printf( "ch[0]=%c et *pc=%c\n", ch[0],*pc); → ch[0]=N et *pc=o
int f(int x) {
    x *= 2;
    return x;
}
int g(int *x) {
    *x += 10;
    return *x;
}
int a = 10, b=f(a);
int c = g(&b);
int d = f(g(&a));
int e = f(f(g(&c)));
printf( "a= %d, b=%d, c=%d, d=%d et e=%d\n",a,b,c,d,e);
// a=20, b=30, c=40, d=40 et e=160
```

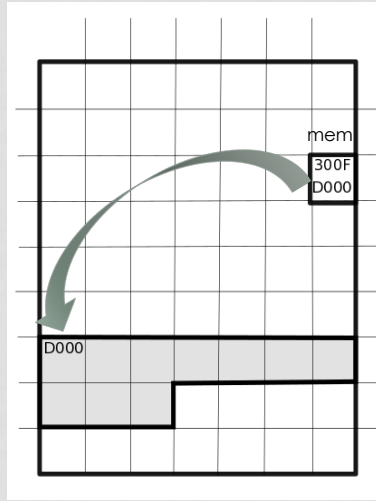
ALLOCATION MÉMOIRE

- Mémoire est allouée lors de la déclaration de la variable
 - Type simple
 - int, float, char, ...
 - Ex : int x=9, float y=0.3
 - Type complexe
 - enregistrement (struct), tableau,...
 - Ex : personne p, char tab[5]
 - Pointeur
 - Ex : int* pInt, float* pFl, personne* pPers, char* pTab



ALLOCATION DYNAMIQUE

- Réserver un espace mémoire à l'exécution du programme
 - Opposer à l'allocation statique
- Sans savoir le type de données et la taille à l'avance
 - Utiliser des pointeurs
- Allouer un espace mémoire à la demande
 - Un type de données
 - Une taille
 - Constante ou via variable
 - Exemple:
 - `int* mem=malloc(5*sizeof(int))`
- Libérer la mémoire
 - Attention aux fuites de mémoire



ALLOCATION DYNAMIQUE

- Syntaxe en C/C++
 - Inclure la bibliothèque : `#include<stdlib.h>`
 - Allouer la mémoire (**M**emory **A**llocation)
 - `void* nom_pointeur = malloc(size_t taille)`
 - `type_donnees* nom_pointeur = (type_donnees*) malloc(size_t taille*sizeof(type_donnees))`
 - `void *calloc (size_t taille, sizeof(type_donnees))`
 - Modifier la taille mémoire allouée
 - `void *realloc (void *nom_pointeur, size_t taille)`
 - Libérer la mémoire
 - `void free(void* nom_pointeur)`

EXEMPLES : ALLOCATION DYNAMIQUE

- Allouer dynamique une variable
 - `int* p = (int *)malloc(sizeof(int)); ⇔ int x;`
 - `*p = 1; ⇔ x = 1;`
- Un tableau 1D de taille n
 - `float* t = (float *)malloc(n * sizeof(float));`
 - `*t = 1; ⇔ t[0] = 1;`
- Une structure de donnée
 - `typedef struct {`
 - `double val;`
 - `int key;`
 - `} element;`
 - `element* t = (element *)malloc(sizeof(element));`
 - `(*t).key = 5; ⇔ t->key = 5;`

ALLOCATION DYNAMIQUE

- Manipuler un pointeur = manipuler la variable allouée dynamiquement
 - `int* p = (int *)malloc(sizeof(int));` //Allouer dynamiquement un int
 - `*p = 1;` //Modifier la case mémoire pointée par p avec 1
- Fuite de mémoire
 - Absence de la désallocation
 - Accès à la mémoire qui ne sont plus référencés
 - Saturation de la mémoire de la machine
- Règles à respecter lors d'allocation dynamique
 - Chaque allocation malloc doit être libéré par free
 - Initialisation à NULL toute déclaration de pointeur
 - Avant le malloc, vérifie que le pointeur est NULL
 - Après le malloc, vérifie que la mémoire est bien alloué
 - Avant le free, vérifie que le pointeur n'est pas NULL
 - Après le free, réinitialise le pointeur à NULL

EXEMPLES : ALLOCATION DES TABLEAUX

```
int nbNotes, i;
char prenom[20];
printf( " saisir le nombre de notes : " );
scanf( " %d ", &nbNotes );
float* notes=malloc(nbNotes*sizeof(float));
for (i=0; i<nbNotes; i=i+1) {
    printf(" saisir le %d ieme note : ", i+1);
    scanf(" %d ", &notes[i]);
}
printf( "saisir prenom : " );
scanf( "%s", prenom );
notes[0]=15.25;
notes[1]=12.5;
printf( "Les notes de %s sont :\n", prenom );
for (i=0; i<nbNotes; i=i+1)
    printf( " %d ", notes[i] );
//calcule la moyenne,...
```

Déclaration des variables

Saisie le tableau

Modification des notes

Affichage

notes + i

EXEMPLES : ALLOCATION DES TABLEAUX

```
int taille, dim1, dim2, i;
float* table1D=malloc(taille*sizeof(float));

float** table2D=malloc(dim1*sizeof(float*));
if(table2D == NULL)
    exit(EXIT_FAILURE);
for(i=0; i<taille1 ; i++)
    table2D[i]=malloc(dim2*sizeof(float));

//Manipulation des tableaux

free(table1D);
table1D=NULL; //Par mesure de sécurité

for(i=0; i < dim1; i++)
    free(table2D[i]);
free(table2D);
table2D= NULL; //Par mesure de sécurité
```

Dimension 1

Dimension 2

dim1=5, dim2=3

Libération de mémoire après utilisation

EXEMPLES : ALLOCATION DES TABLEAUX

- Fonction alloue un tableau


```
float * cree_tableau(int nb_elements) {
    float* T = malloc(nb_elements*sizeof(float));
    if(T == NULL)
        exit(EXIT_FAILURE);
    return T;
}
float* tab = cree_tableau(10);
```
- Fonction libère un tableau


```
void libere_tableau(float* T) {
    free(T);
    T = NULL;
}
libere_tableau(tab);
```

EXEMPLES : ALLOCATION DES TABLEAUX

- Fonction alloue un tableau 2D


```
float** cree_tableau(int taille1, int taille2) {
    float ** T = malloc(taille1*sizeof(float*));
    if(T == NULL)
        exit(EXIT_FAILURE);
    for(int i = 0; i < taille1 ; i++)
        T[i] = malloc(taille2*sizeof(float));
    return T;
}
float** tab = cree_tableau(10, 20);
```
- Fonction libère un tableau 2D


```
void libere_tableau(double** T, int taille1, int taille2) {
    for(int i = 0; i < taille1; i++)
        free(T[i]);
    free(T);
    T = NULL;
}
libere_tableau(tab,10, 20);
```


ALLOCATION DYNAMIQUE POUR LE PROJET

- Mémoire dynamique pour chaque pièce

```
piece* p1=malloc(sizeof(piece));
piece* p2=malloc(nbPiece*sizeof(piece));
```

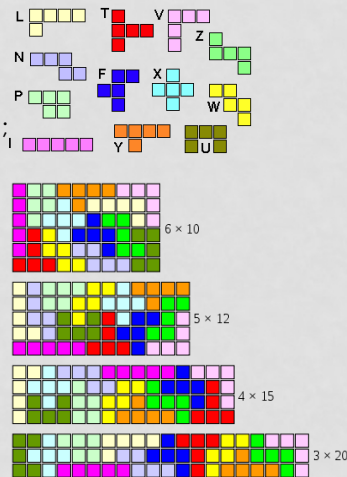
- Mémoire dynamique pour la grille pièce

```
grille* g=malloc(sizeof(grille));
```

- Mémoire dynamique pour l'image des carrées (en SDL)

```
SDL_Surface* temp = SDL_LoadBMP(img);
```

- Et ???



CHAINE DE CARACTÈRES

- Une chaîne de caractère est aussi un tableau
 - `char* chaine = "une chaîne de caractères"`
- Opérations sur les pointeurs
 - Addition, soustraction, incrémentation, décrémentation, ...
- Fonctions dédiées aux chaînes de caractères avec **#include <string.h>**
 - Longueur : `size_t strlen(const char * chaine);`
 - Comparaison : `int strcmp(const char * c1, const char * c2);`
 - Copie : `char * strcpy(char * dst, const char * src);`
 - Concaténation : `char * strcat(char * dst, const char * src);`
 - Recherche : `char * strchr(const char * chaine, int carac);`

POINTEUR SUR FONCTIONS

- Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction
- Déclaration
 - La fonction : `type fonction(type 1, ..., type n);`
 - Le pointeur : `type (*ptr_fct)(type 1, ..., type n);`
- Initialisation le pointeur avec **&**:
 - `ptr_fct = &fonction`
- Déréférencement avec *****
- Exemple :

```
int somme(int a, int b) { return a + b; }
```

```
void main(void) {
    int (*pt)(int, int) = &somme; // Déclaration + initialisation
    printf("%d.\n", (*pt)(3,5));    // Déréférencement
}
```

↓ somme(3,5)

EXEMPLES : POINTEUR SUR FONCTIONS

- Passer une fonction comme argument

```
int somme(int, int);
int produit(int, int);
int operateur_binaire(int, int, int (*)(int, int));
```

```
int somme(int a, int b) { return(a + b); }
int produit(int a, int b) { return(a * b); }
```

```
int operateur_binaire(int a, int b, int (*)(int, int)) {
    return((*f)(a, b));
}
```

```
void main(void) {
    printf("%d\n", operateur_binaire(3, 5, somme));
    printf("%d\n", operateur_binaire(3, 5, produit));
}
```

EXEMPLES : POINTEUR SUR FONCTIONS

- Passer une fonction comme argument

```
int somme(int, int);
int produit(int, int);
int operateur_binaire(int, int, int (*)(int, int));

int somme(int a, int b) { return(a + b); }
int produit(int a, int b) { return(a * b); }

int operateur_binaire(int a, int b, int (*)(int, int)) {
    return((*f)(a, b));
}

void main(void) {
    printf("%d\n", operateur_binaire(3, 5, somme));
    printf("%d\n", operateur_binaire(3, 5, produit));
}
```

EXEMPLES : POINTEUR SUR FONCTIONS

- Retourner un pointeur de fonction

```
void affiche_paire(int a) { printf("%d est pair.\n", a); }
void affiche_impair(int a) { printf("%d est impair.\n", a); }
void (*affiche_valeur(int a))(int) {
    if (a % 2 == 0)
        return &affiche_paire;
    else
        return &affiche_impair;
}

void main(void) {
    void (*pf)(int);
    pf = affiche_valeur(2); ⇔ (*pf)(2);
    pf = affiche_valeur(3); ⇔ (*pf)(3);
}
```

EXEMPLES : POINTEUR SUR FONCTIONS

- Retourner un pointeur de fonction

```
void affiche_paire(int a) { printf("%d est pair.\n", a); }
void affiche_impair(int a) { printf("%d est impair.\n", a); }
void (*affiche_valeur(int a))(int) {
    if (a % 2 == 0)
        return &affiche_paire;
    else
        return &affiche_impair;
}

void main(void) {
    void (*pf)(int);
    pf = affiche_valeur(2); ⇔ (*pf)(2);
    pf = affiche_valeur(3); ⇔ (*pf)(3);
}
```

RÉCAPITULATIFS

- Mémoire sur ordinateur
- Structure de programme dans la mémoire
- Notion de pointeur
 - Sur les variables : adresse (&) et valeur (*)
 - Sur les fonctions
- Allocation dynamique de mémoire
 - Un tableau, des structures, les listes chaînées, ...
- La visibilité des variables locales et globales
- Le passage des paramètres dans les fonctions
 - Passage par valeur et par référence
- Les constantes dans le programme

CONTENUE DU COURS

- Rappels
 - Fonctions et procédures
 - Structures de données
 - Enregistrements, tableaux, liste enchainé, Pile(FIFO), file(FILO), ...
- Compilation avec Makefile/Cmake et la bibliothèque SDL
- Gestion de mémoire
 - Représentation de mémoire
 - Pointeurs et allocation dynamique
- **Gestion de fichiers**
 - **Lecture et écriture**
- **Gestion des entrées et sorties**
 - **Ecran, souris et clavier**
- Directives au préprocesseur
- Structures de données et algorithmes de graphes
 - Parcours de graphe, arbre couvrant, ...