Structures Dynamiques des Données et Pointeurs: C

Pr. Abdelali El Gourari

Caractèristique du C

- Structuré
- Modulaire: peut être **découpé en modules** qui peuvent être compilés séparément
- Universel: n'est pas orienté vers un domaine d'application particulier
- Typé: tout objet C doit être **déclaré** avant d’être utilisé
- Portable: sur **n'importe quel système** en possession d'un compilateur C

Un long fleuve tranquille

C
fichier

Compilateur C

Code
assembleur

Assembleur

Code
objet

Linker

Code
executable

Fichier C (extension .c)

```
/* exemple de programme C :
- somme des nb de 1 à 10 et affichage
de la valeur*/
 #include <stdio.h>  0
 int main (void)     1
 {
    int somme; int i;      2
    somme = 0;  3
    for (i = 1;  i <= 10; i++)
4  {
        somme = somme + i;
     }
   printf ("%d\n", somme);  5
    somme = 0;
}
```

En C le programme principal s'appelle toujours *main* 1

déclarations de variables de type entier (cases mémoire pouvant contenir un entier) 2

instruction d'affectation de valeur à la variable *somme* 3

instructions exécutées en séquence

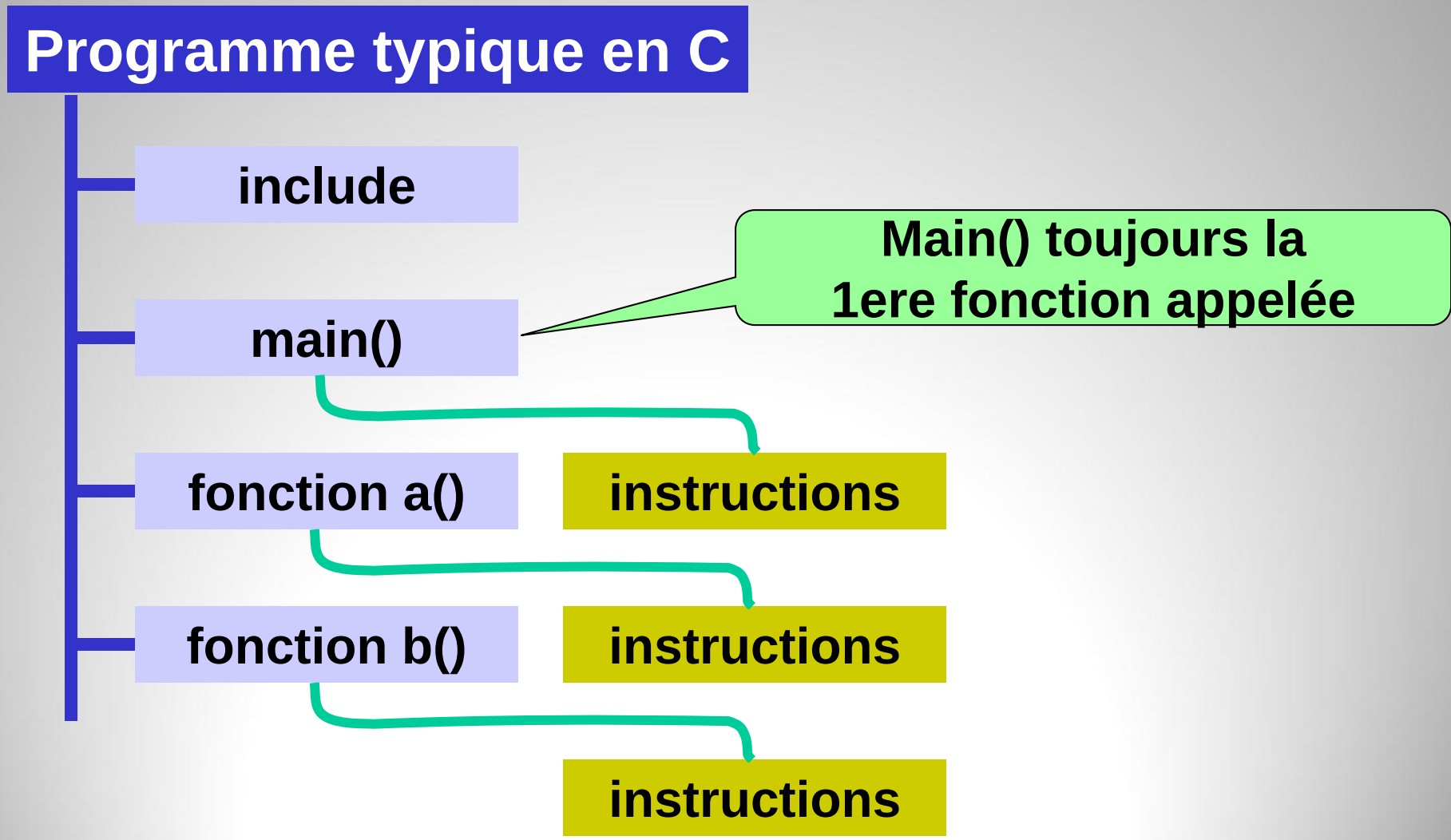
l'instruction entre accolades est exécutée pour les valeurs de *i* allant de 1 à 10 4

affiche à l'écran la valeur de l'entier contenu dans *somme* 5

Ecrire le programme suivant :

```
#include <stdio.h>
int main(void) {

    int a= 257, b = 381;

    if (a > 0 && b > 0) {
        printf("  PGCD(%3d,%"d)\n",a,b);

        while (a != b) {
            if (a < b)
                b = b-a;
            else
                a = a-b;
        printf("=PGCD(%3d,%3d)\n",a,b);
        }
        printf("=%d\n",a);
    }
    return 0;
}
```

Anatomie


Programme typique en C
include
main()
Main() toujours la
1ere fonction appelée
fonction a()
instructions
fonction b()
instructions
instructions

Commandes simples de compilation

- Prétraitement, compilation et édition de liens : *–Wall –g pgcd.o –lm –o pgcd*
- l'option *–Wall* demande une compilation avec des diagnostics sur la propreté du code
- l'option *–g* demande que la *table des symboles* soit ajoutée à l'exécutable
- l'option *–lm* demande de lier la librairie mathématique
- l'option *–o pgcd* demande que le résultat (l'exécutable) soit nommé pgcd au lieu de a.out
- Le programme est à lancer avec ./pgcd

Types d’instruction en C

- Déclarations des variables
- Assignations
- Fonctions
- Contrôle

Types de données et de variables

- Déclaration des variables
  - int y;
  - char yesno, ok;
  - int ordered = 1, onhand = 0;
  - float total = 43.132;
  - char *cptr = NULL;

Opérateurs et expressions

- Opérateurs à un paramètre:
  - - change le signe de la variable
  - ~complément à 1
  - * *« indirection » (pointeurs)*
    - *value = *salary; /* contenu pointé par salaire */*
  - &adresse
  - ++/-- incrémentation/décrémentation
  - sizeof()

Opérateurs et expressions

- Opérateurs arithmétique:
  - *,/,+,-
  - % modulo
- Opérateurs sur bits:
  - <<,>> décalage à gauche ou à droite
    - status = byte << 4;
  - &et
  - | ou
  - ^ ou exclusif

Opérateurs et expressions

- Opérateurs relationnels:
  - <,>,<=,=>
- Opérateurs d’égalité:
  - ==, !=
- Opérateurs logiques:
  - && et
  - || ou

Opérateurs et expressions

- Opérateur conditionnel:
  - result = mode > 0 ? 1 : 0;
  - if mode>0 then result=1 else result=0.

- Opérateurs d’assignation:
  - =, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=

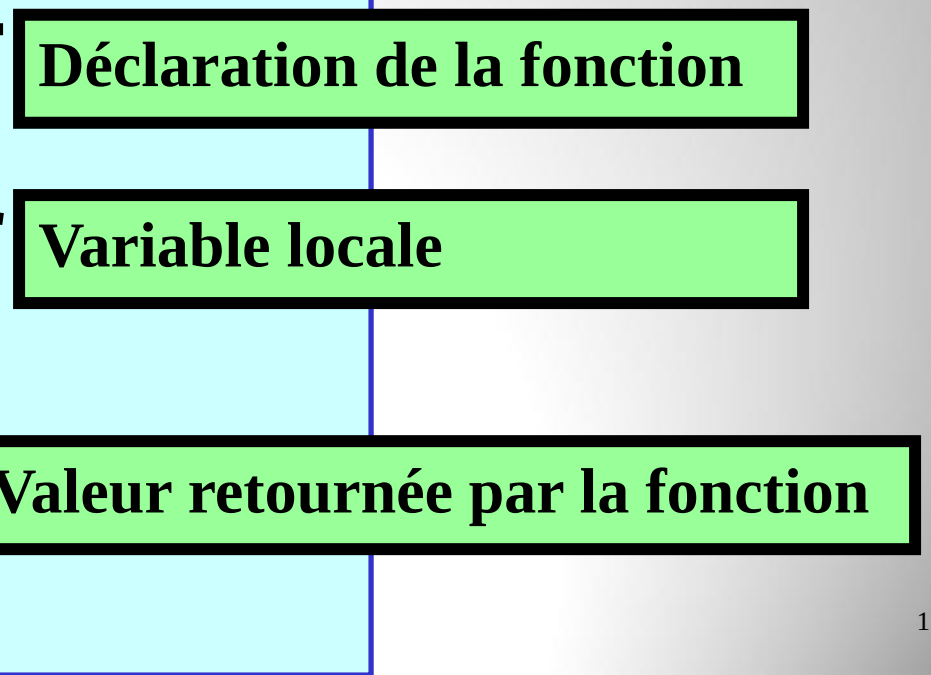
Fonctions en C

- Plusieurs fonctions pré-définies:
  - printf(), sin(), atoi(), …
- Le prototype de ces fonctions sont dans fichiers d'entête (header file)
  - printf() dans stdio.h
  - sin() dans math.h

Fonctions en C

- Bien sûr, nous pouvons écrire nos propres fonctions.

```
/* Routine de calcul du maximum */
int imax(int n, int m)
{
   int max;

   if (n>m)
         max = n;
   else
         max = m;
   return max;
}
```



Fonctions en C

- Fonctions **sans arguments** et ne retournant pas de valeur.
  - void fonction(void)

- Fonctions **avec arguments** ne retournant pas de valeur.
  - void fonction(int x, int y, char ch)

Fonctions en C

- Les fonctions exigent la déclaration d'un prototype avant son utilisation:

```
/* Programme principal */
#include <stdio.h>
int imax(int,int);
main()
{   …   }

int imax(int n, int m)
{   …   }
```

Prototype de la fonction

La fonction est définie ici

Boucle « for »

```
/* Boucle for */
#include <stdio.h>
#define NUMBER 22
main()
{
   int count, total = 0;

   for(count =1; count <= NUMBER; count++, total += count)
         printf("Hello !!!\n");
   printf("Le total est %d\n", total);
}
```

Initialisation

Condition de fin de boucle

Incrémentation et autres fonctions

Boucle « while »

```
/* Boucle while */
#include <stdio.h>
#define NUMBER 22
main()
{
   int count = 1, total = 0;

   while(count <= NUMBER)
   {
      printf("Vive le langage C !!!\n");
      count++;
      total += count;
   }
   printf("Le total est %d\n", total);
}
```

Initialisation

Condition de fin de boucle
(boucle tant que vrai)
(*boucle faite que si vrai*)

Incrémentation

Boucle « do while »

```
/* Boucle do while */
#include <stdio.h>
#define NUMBER 22
main()
{
   int count = 1, total = 0;

   do
   {
      printf("Vive le langage C !!!\n");
      count++;
      total += count;
   }  while(count <= NUMBER);
   printf("Le total est %d\n", total);
}
```

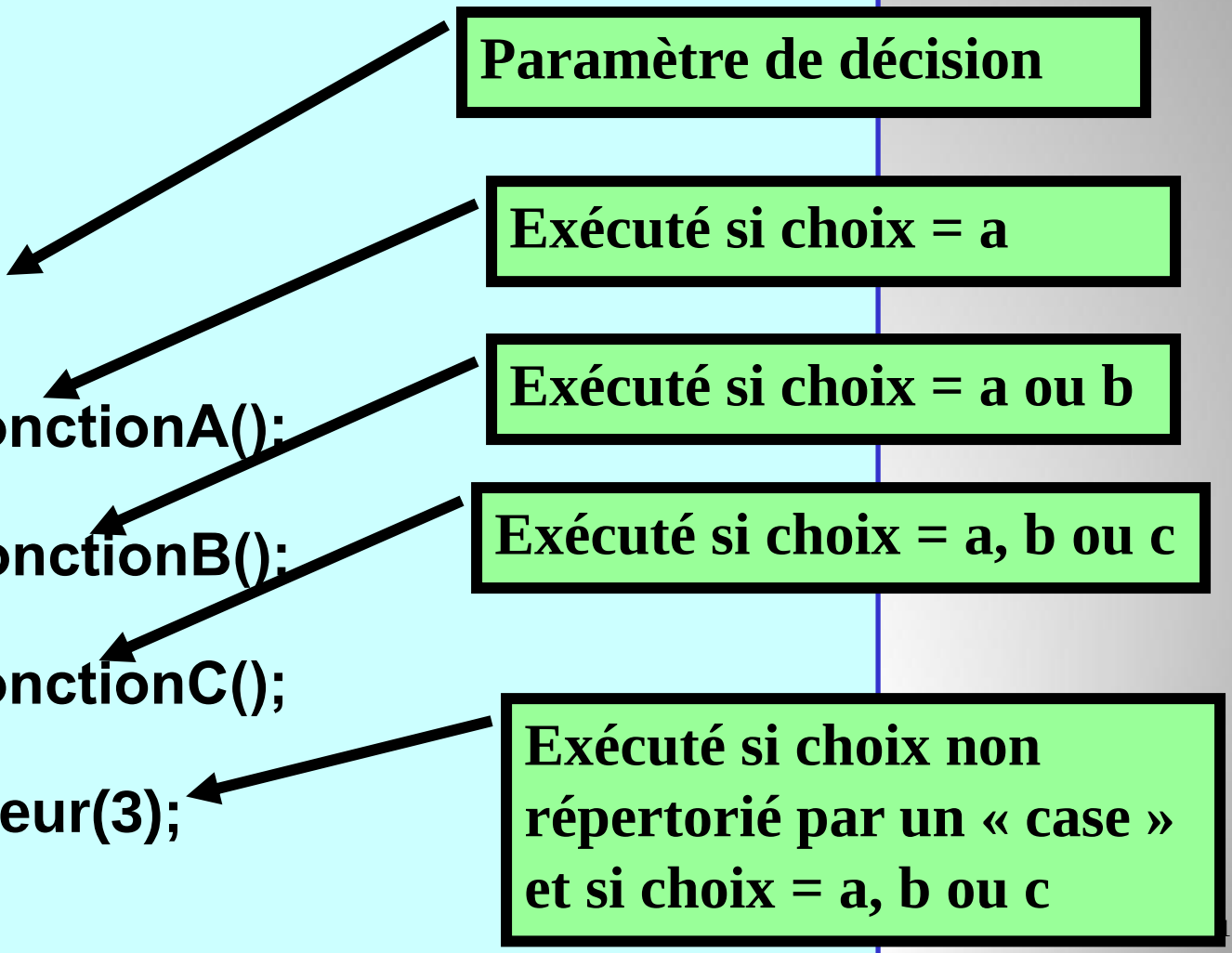
Initialisation

Incrémentation

Condition de fin de boucle
(boucle tant que vrai)
(*boucle faite au moins 1 fois*)

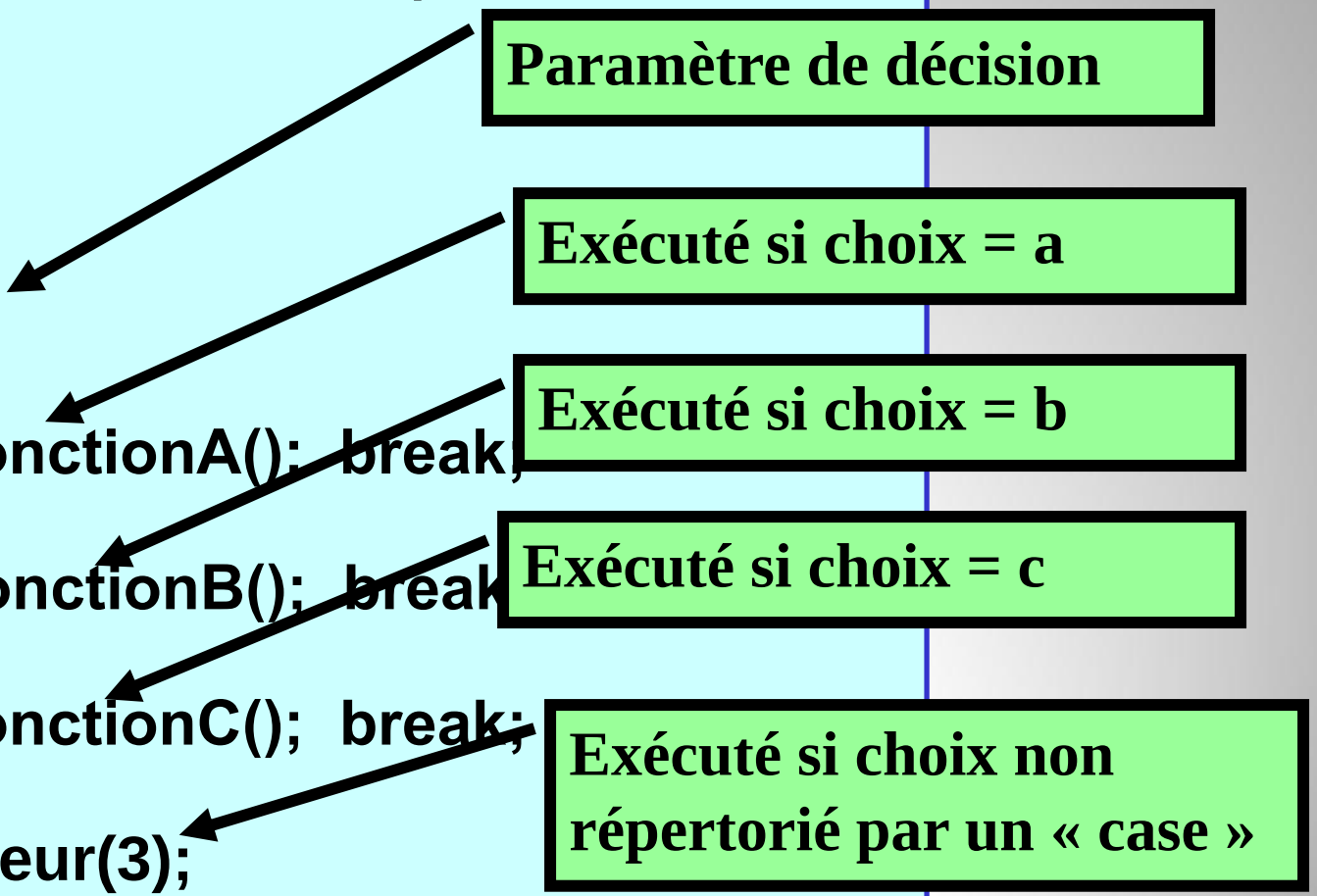
Choix multiple: « switch case »

```
/* Utilisation de switch case */
main()
{
   char choix;
   …
   switch(choix)
   {
      case ‘a’ :  fonctionA();

      case ‘b’ :  fonctionB();

      case ‘c’ :  fonctionC();

      default : erreur(3);
    }
}
```



Effet du « break »

```
/* Utilisation de switch case */
main()
{
   char choix;
   …
   switch(choix)
   {
      case ‘a’ :  fonctionA();  break;

      case ‘b’ :  fonctionB();  break;

      case ‘c’ :  fonctionC();  break;

      default : erreur(3);
    }
}
```



Les pointeurs

- Un pointeur est une **variable spéciale**
- **Une variable:** est caractériser par **Adresse**, **Nom** et **Valeur**.

Adresse	***1A00***
Nom	A
Valeur	25

- Un **pointeur** contient l'adresse d'une autre variable.

Adresse	**1F03**
Nom	p
Valeur	***1A00***

- Déclaration d'un pointeur:
  - **Type_variable_pointée *Pointeur;**

int *P; /*p peut contenir l'adresse d'une variable de type entier*/

char *p; /*p peut contenir l'adresse d'une variable de type caractère ou chaîne de caractères*/

Les pointeurs

- Remarque :

Lorsqu'un pointeur ne contient aucune adresse valide, il est égal à **NULL** (Pour utiliser cette valeur, il faut inclure le fichier **stdio.h** dans le programme).

- OPERATEURS **&** ET *****

Le langage C met en jeu deux opérateurs utilisés lors de l'usage de pointeurs. Il s'agit des opérateurs **&** et *****.

- ***& variable signifie adresse de variable.***
- **** pointeur signifie le contenu de l'adresse référencée par pointeur.***

Les pointeurs

- **Adressage direct**: Accès au contenu d'une variable par le nom de la variable

int A;

A=25;

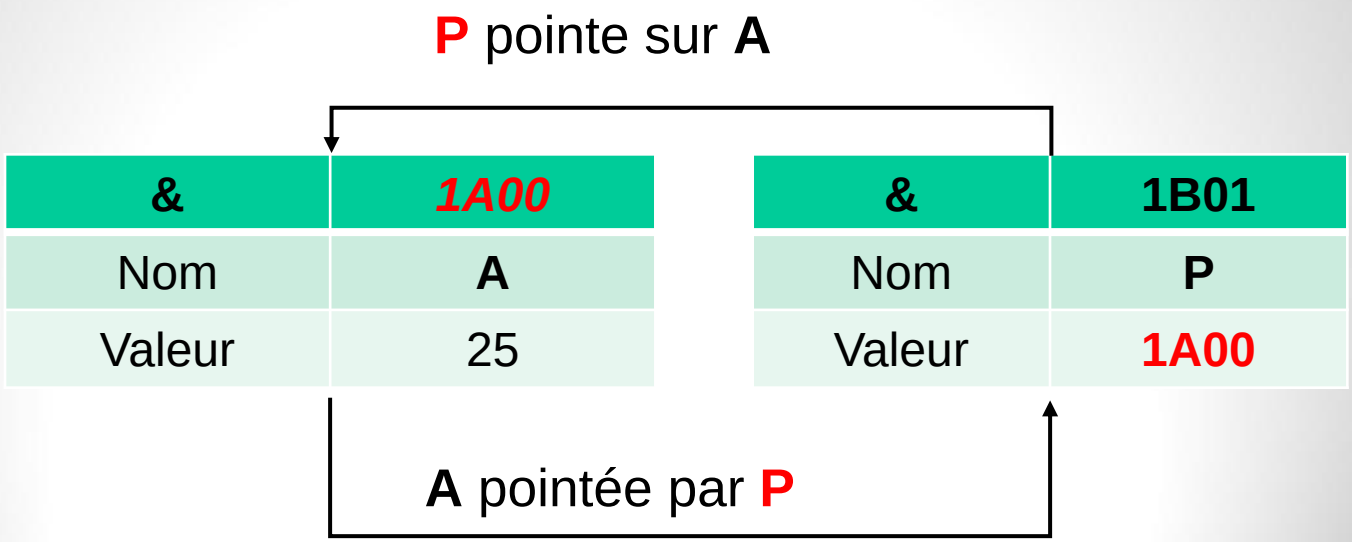
Adresse	***1A00***
Nom	A
Valeur	25

- **Adressage indirect**: Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

int A;

int* P;

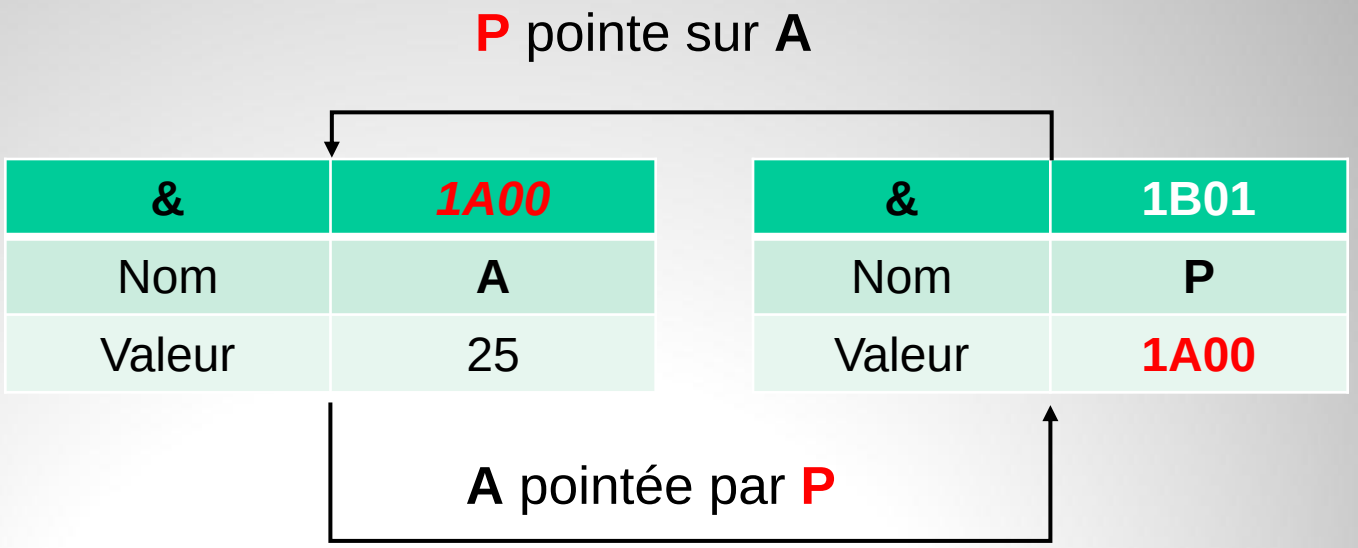
P=&A;



Les pointeurs

***P=25;**

***P↔A (la valeur de A);**



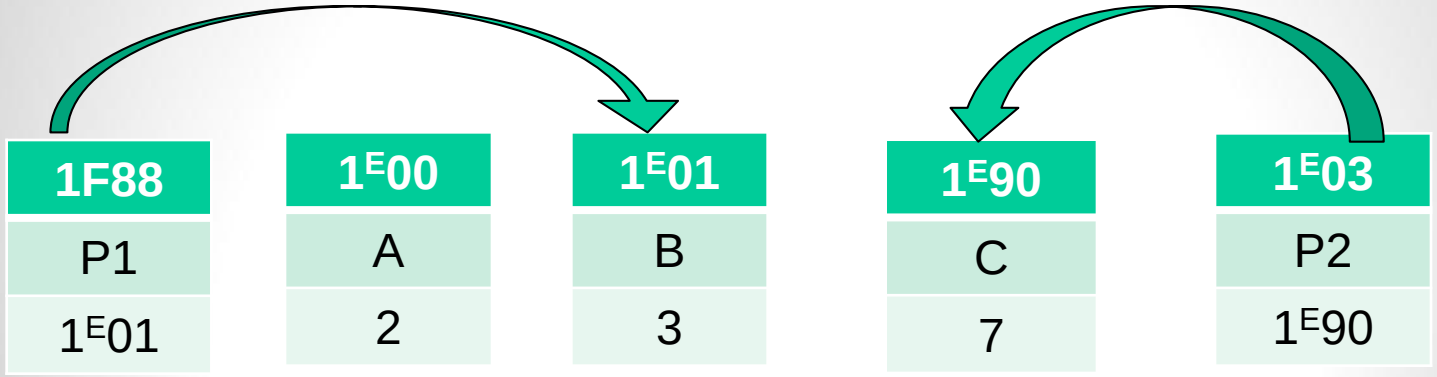
scanf(“%d”,&A); ↔ scanf(‘%d’ ,P);

printf(“%d”,A); ↔ printf(‘%d’ ,*P);

Les pointeurs

Exemple:

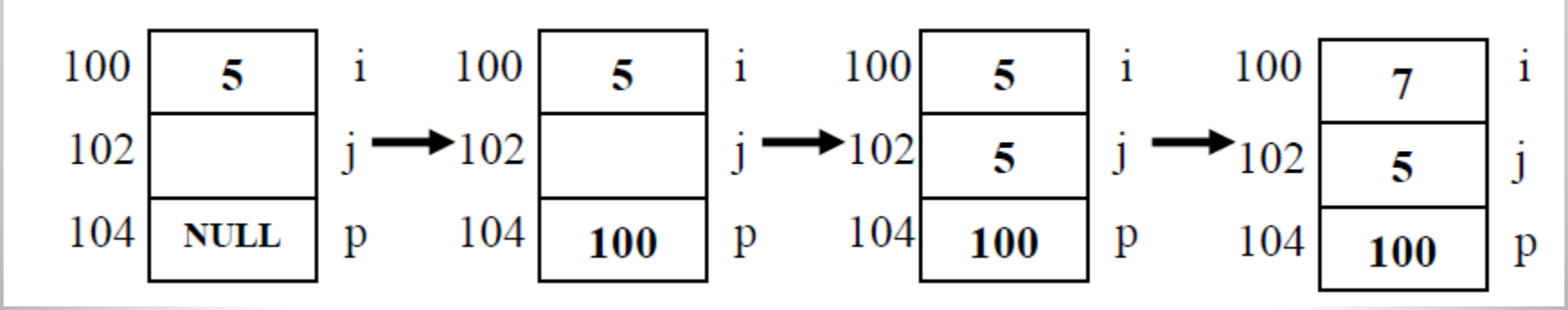
```
Void main()
{
Int A=2, B=3, C=7;
Int *P1, *P2;
P1=&A;
P2=&B;
B=*P1+*P2;
P1=&B;
printf("%d----------%X",*P1,P2);

}
```



Les pointeurs

```
#include <stdio.h>
main( )
{
int i,j;          /*i et j des entiers*/
int *p;           /*p pointeur sur un entier*/
i=5;              /*i reçoit 5*/
p=&i;             /*p reçoit l'adresse de i*/
j=*p;             /*j reçoit 5 : contenu de l'adresse p*/
*p=j+2;           /* le contenu de l'adresse p devient 7 donc i aussi devient
7*/
}
```

Représentation mémoire(supposant que i, j et p se trouvent respectivement aux adresses 100, 102 et 104)



Les pointeurs: Exercices

Exercice 1:
Ecrire un programme C qui utilise la notion de pointeur pour lire deux entiers et calculer leur somme.

Exercice 2:
Ecrire un programme C qui utilise la notion de pointeur pour la permuter le contenu de deux variables de type char.

Les pointeurs: Exercices

Exercice 1:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
	int a,b,s;
	int *pa,*pb,*ps;
		pa = &a;
		pb = &b;
		ps = &s;
	printf("donnez deux entiers:\n");
	scanf("%d%d",pa,pb);
		*ps = *pa + *pb;
	printf("la somme est %d.\n",s);
	system("pause");
	return 0;
}
```

Les pointeurs: Exercices

Exercice 2:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
        char a,b,       *pa,    *pb;
        char tmp;       pa = &a;        pb = &b;
    printf("Entrez le premier charactere (a): ");
    scanf("%c",pa);
    getchar();
    printf("Entrez le deuxieme charactere (b): ");
    scanf("%c",pb);
    printf("a = %c et b = %c.\n",a,b);
        tmp = *pa;
        *pa = *pb;
        *pb = tmp;
    printf("a = %c et b = %c.\n",a,b);
    system("pause");
    return 0;
}
```

Les pointeurs et les tableaux

Int A[5];

&A[0] ⟷ A

&A[1] ⟷ A+1

&A[i] ⟷ A+i

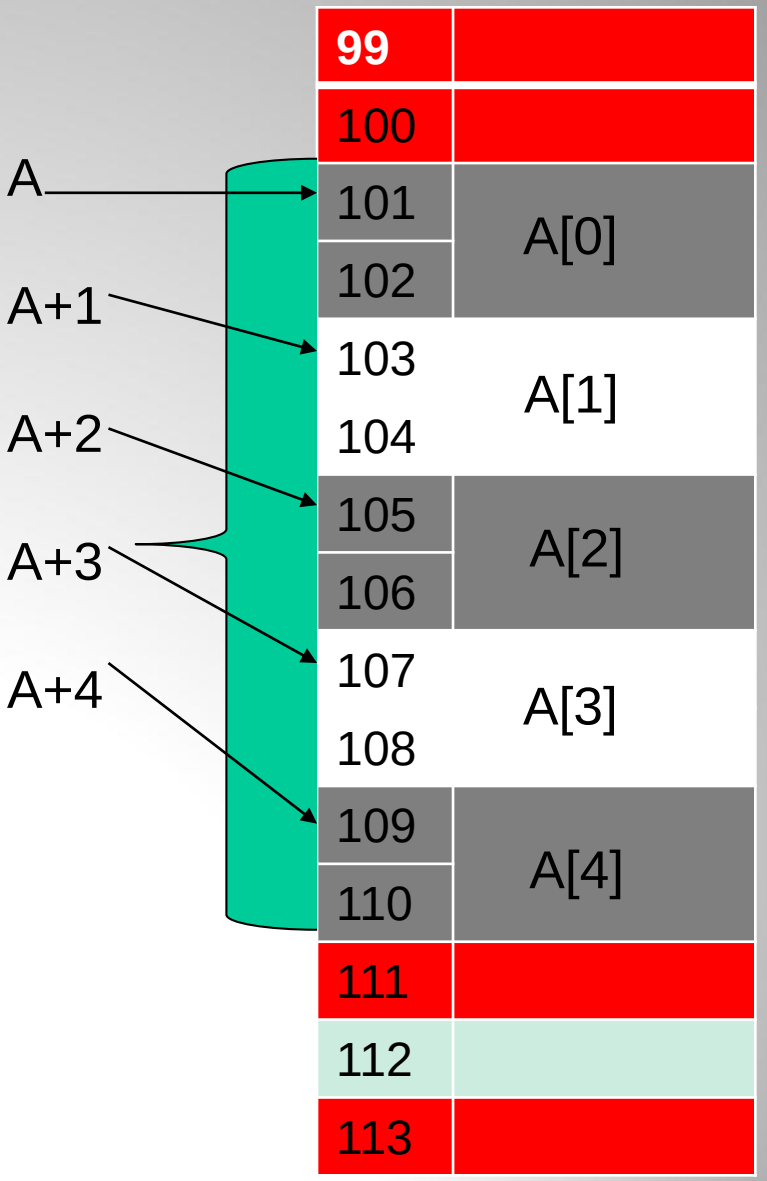
int B[5];

A[0] ⟷ *A

A[1] ⟷ *(A+1)

A[i] ⟷ *(A+i)

A=B ← ???? → **A=A+1**




```
for(i=0; i<5; i++)
Scanf(‘’%d’’, &A[i]);
```

```
for(i=0; i<5; i++)
Scanf(‘’%d’’, A+i);
```

Les pointeurs et les tableaux

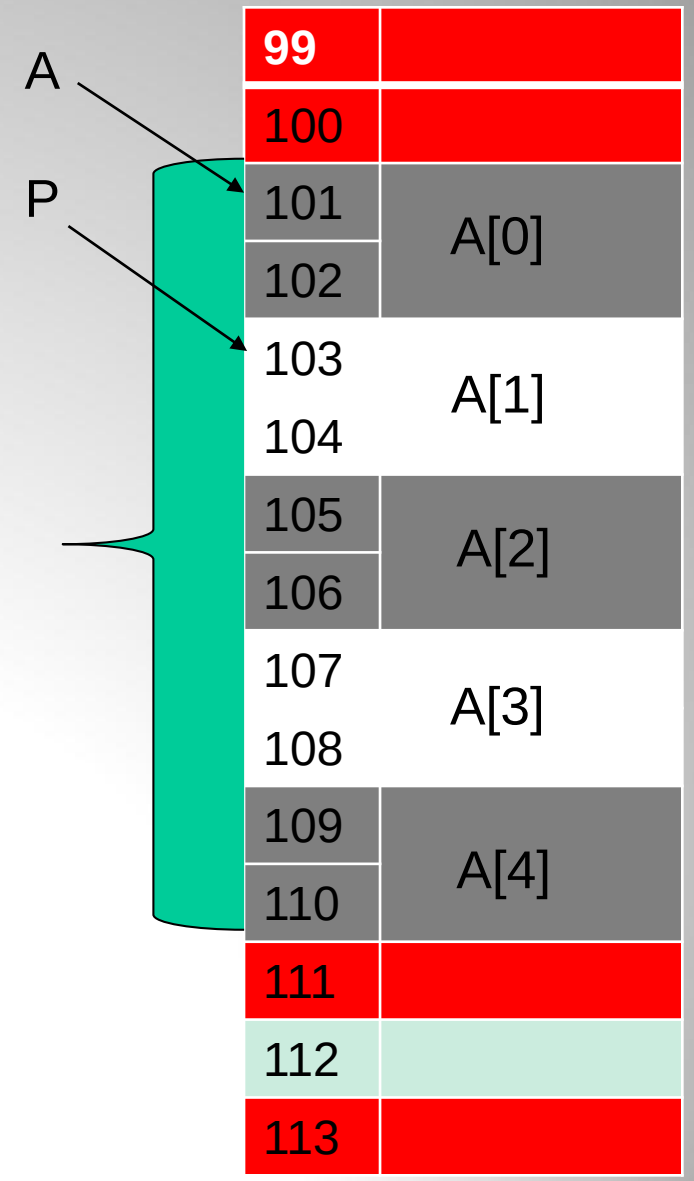
Un pointeur peut être **déplacé** d'une adresse à une autre au moyen des opérateurs ++ et --.
L'unité d'incrémentation (ou de décrémentation) d'un pointeur est toujours la **taille de la variable pointée**.

Int A[5];
Int *P = NULL;
P = A;
*P = 20;⟷*A=20; ⟷ A[0] = 20;

P++; ⟷ P=P+1; ⟷ A=A+1;
*P=30;

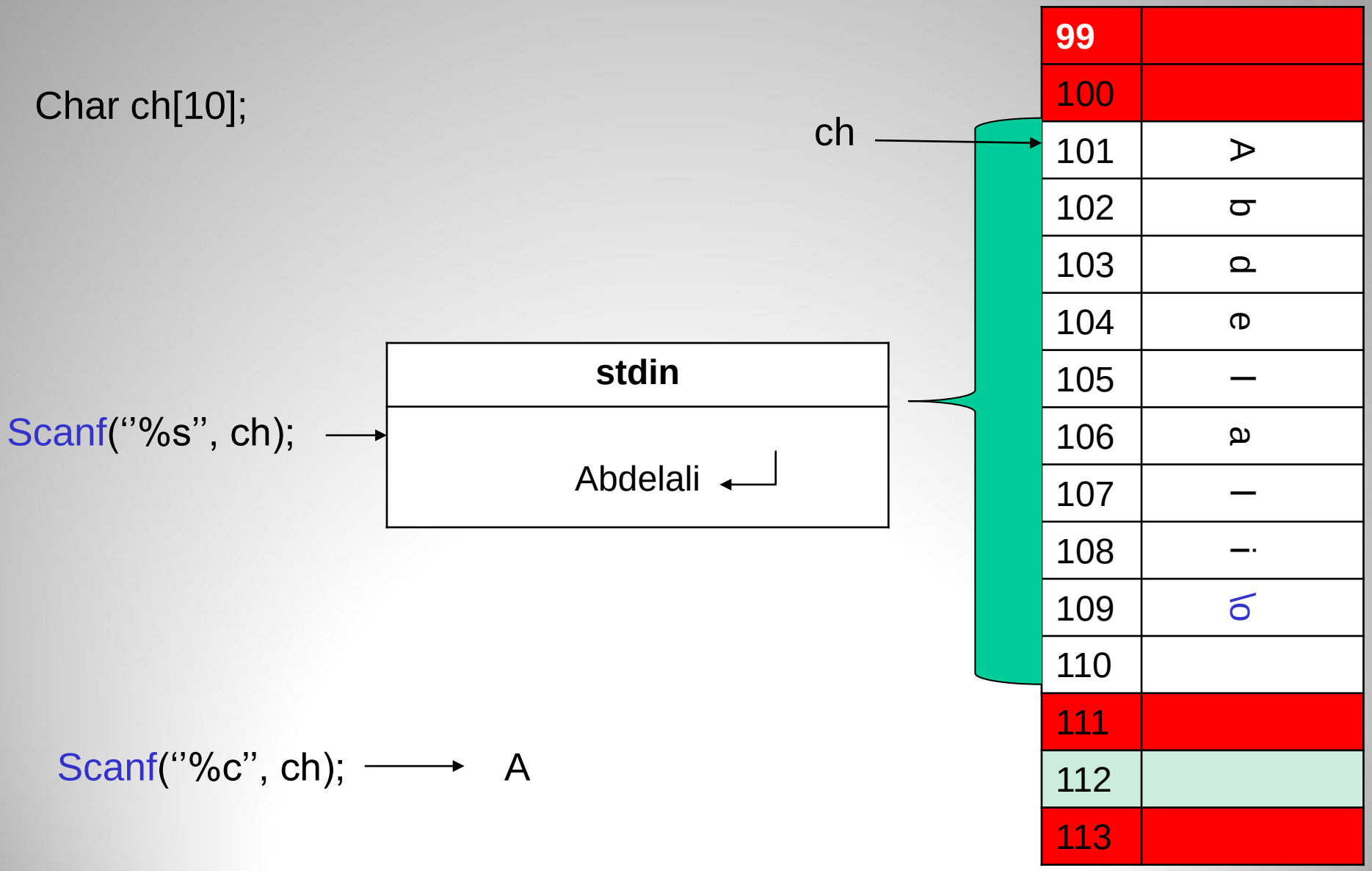
for(P=A; P<A+5; P++)

Scanf(‘’%d”, P);



Les pointeurs et les tableaux

Char ch[10];



Les pointeurs et les tableaux

Soit **P** un pointeur qui **pointe** sur un tableau **A:**

Int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

Int *P; P=A;

Quelles valeurs ou adresses fournissent ces expressions:

- *P+2
- *(P+2)
- &A[4] – 3
- A+3
- &A[7] – P
- P + (*P - 10)
- *(P + *(P+8) - A[7])

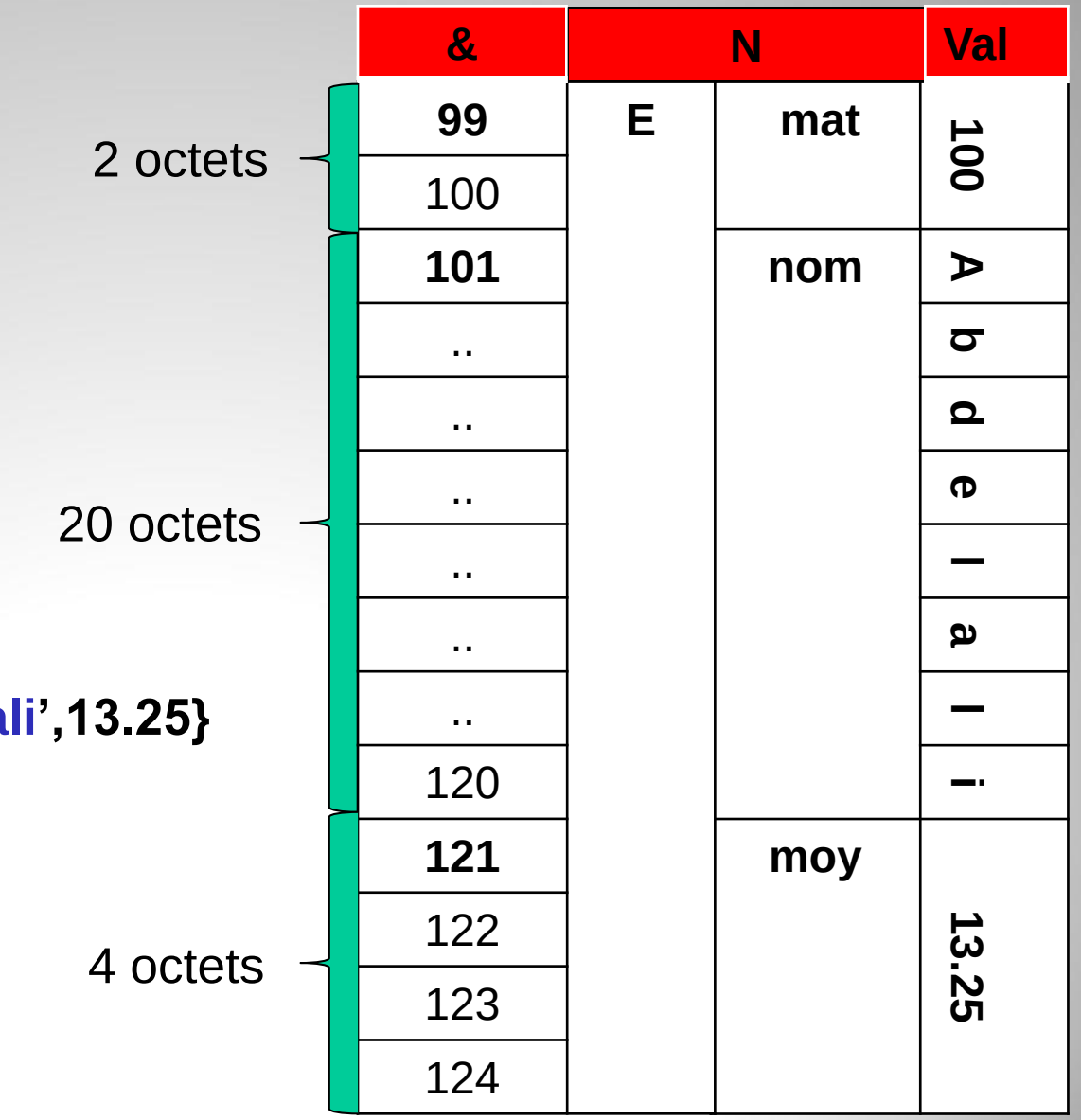
Les pointeurs et les tableaux : Exercices

Exercice 1:
Ecrire un programme C qui remplit un tableau d'entiers et calcule la somme de ses éléments en utilisant un pointeur.

```
#include<stdio.h>
#include<stdlib.h>
int main() {
	int t[50];
	int n,
	*p,som=0;
		printf("Donnez le nombre d'elements: ");
		scanf("%d",&n);
	for(p=t;p<t+n;p++)
		scanf("%d",p);
	for(p=t;p<t+n;p++)
		som = som + *p;
		printf("%d\n",som);
		system("pause");
return 0;
}
```

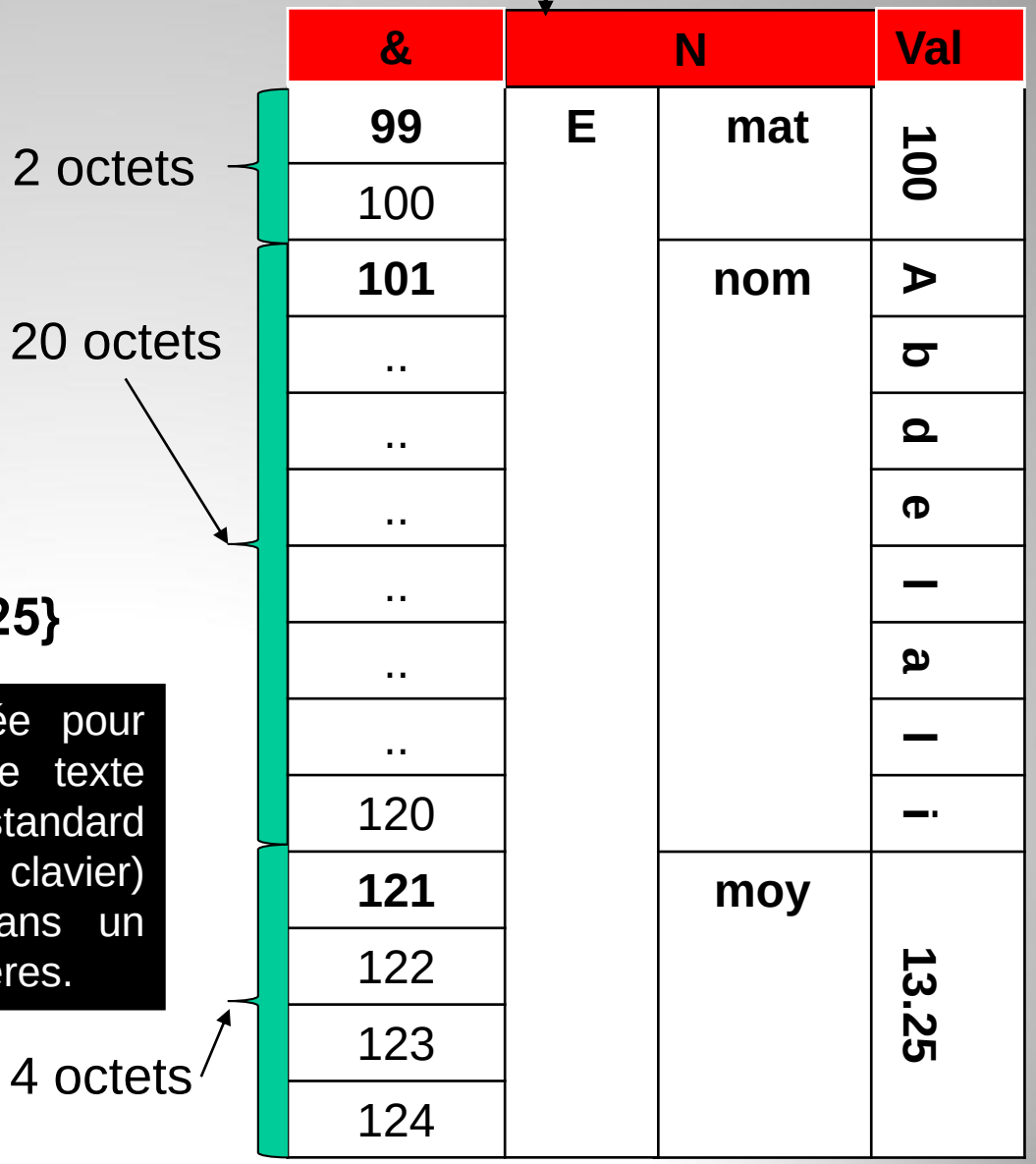
Les pointeurs et les structures

```
Typedef struct etudiant
    {
        int mat;
        char nom[20];
        float moy;
    } etudiant;
    void main()
    {
    etudiant E={100, 'Abdelali',13.25}
```



Les pointeurs et les structures

&	80
N	P
Val	99




```
void main()
{
etudiant E={100, 'Abdelali',13.25}
etudiant *P;
P = &E;
E.mat = 20;
(*P).mat = 20;
P→mat = 20;
gets(P→nom);
scanf('%f',&P→moy);
```

Gets() Est utilisée pour lire une ligne de texte depuis l'entrée standard (généralement le clavier) et la stocker dans un tableau de caractères.

Exercice:

Écrire un programme en langage C qui gère les informations d'un étudiant à l'aide d'une structure.

Indications :

•Définir une structure nommée **Etudiant** qui contient les informations suivantes :

- Nom
- Prénom
- Âge
- Matricule

Écrire une fonction pour saisir les informations d'un étudiant à partir de l'entrée standard (clavier).

Écrire une fonction pour afficher les informations d'un étudiant.

Dans la fonction principale :

- Déclarer une variable de type **Etudiant**.
- Appeler la fonction pour saisir les informations de l'étudiant.
- Appeler la fonction pour afficher les informations saisies.

Exercice:

```
#include <stdio.h>

// Définition de la structure Etudiant
typedef struct Etudiant
{
    char nom[50];
    char prenom[50];
    int age;
    int matricule;
} Etudiant;
```

Exercice:

```
// Fonction pour saisir les informations d'un étudiant
void saisirEtudiant(Etudiant *etudiant) {
        printf("Saisir le nom de l'etudiant : ");
        scanf("%s", etudiant->nom);
        printf("Saisir le prenom de l'etudiant : ");
        scanf("%s", etudiant->prenom);
        printf("Saisir l'age de l'etudiant : ");
        scanf("%d", &etudiant->age);
        printf("Saisir le matricule de l'etudiant : ");
        scanf("%d", &etudiant->matricule); }
```

Exercice:

```
// Fonction pour afficher les informations d'un étudiant
void afficherEtudiant(const Etudiant *etudiant) {
    printf("\nInformations de l'etudiant :\n");
    printf("Nom : %s\n", etudiant->nom);
    printf("Prenom : %s\n", etudiant->prenom);
    printf("Age : %d\n", etudiant->age);
    printf("Matricule : %d\n", etudiant->matricule);
}

int main() {
    // Déclaration d'une variable de type Etudiant
    Etudiant etudiant;

    // Appel de la fonction pour saisir les informations de l'étudiant
    saisirEtudiant(&etudiant);

    // Appel de la fonction pour afficher les informations de l'étudiant
    afficherEtudiant(&etudiant);

return 0;
```

Exercice:

Définissez une structure **Point** représentant un point dans un espace 2D (avec les coordonnées x et y).
Ensuite, créez un tableau de 5 points et écrivez une fonction pour calculer **la somme** et **la moyenne** des coordonnées x et y de ces points.

Exercice :

Définissez une structure **Employe** qui contient les informations suivantes :

nom,
prénom,
numéro d'employé
salaire.

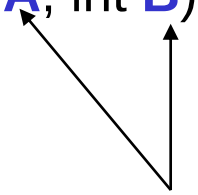
Écrivez une fonction pour afficher les détails d'un employé et une autre fonction pour augmenter son salaire de 10%.

Les pointeurs et les fonction (passage par valeur et passage par adresse)

Paramètres formels:

- Qui figurent dans l'entête de la définition d'une fonction.
- Sont utilisés dans les instructions de la fonction.
- Ils correspondent à des variables locales.

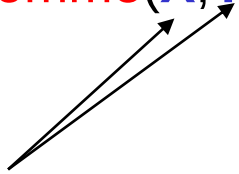
```
int somme(int A, int B)
{
return A+B;
}
```



Paramètres réels:

- Qui figurent dans l'instruction d'appel de la fonction.

```
Void main()
{
Int X= 8, Y=6;
Printf("La somme est %d", somme(X,Y));
}
```



Les pointeurs et les fonction (passage par valeur et passage par adresse)

Il existe deux manières de transmettre des paramètres aux fonctions

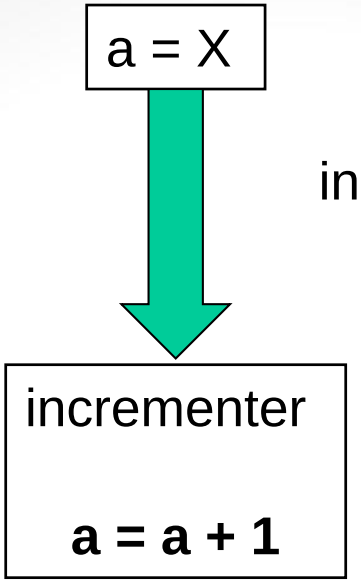
Passage par valeur:	**Passage par adresse:**
Une **copie** de la valeur des paramètres réels est transmise aux paramètres formels respectifs.	L'adresse des paramètres réels est transmise à des paramètres formels.

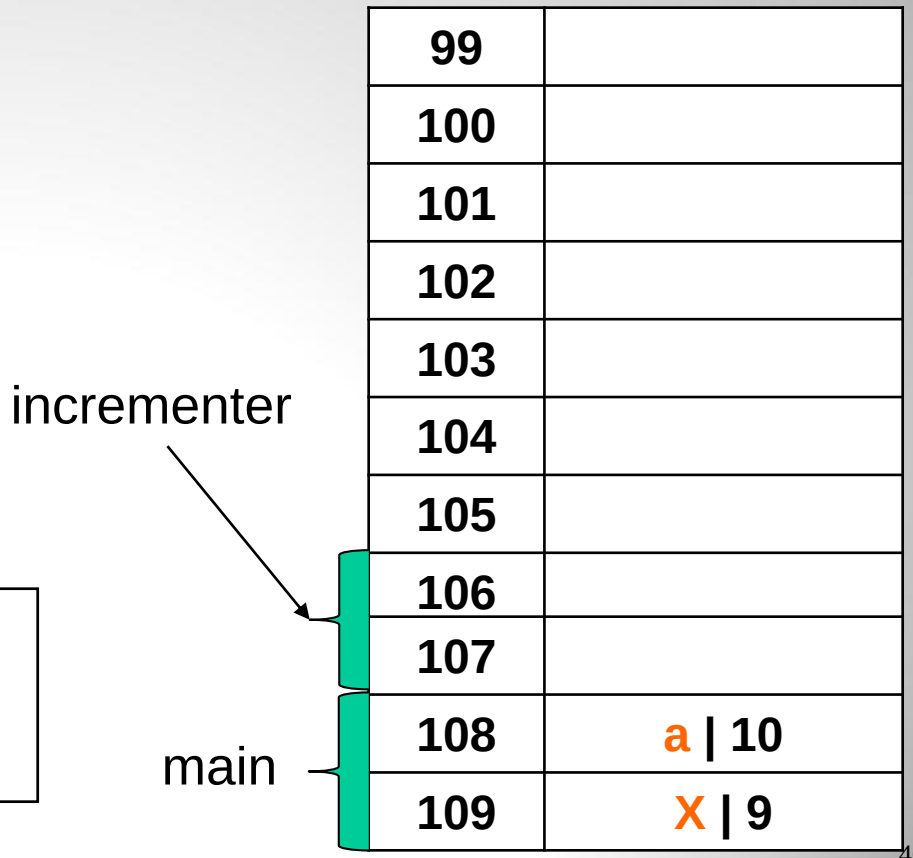
Les pointeurs et les fonction (passage par valeur et passage par adresse)

Exemple:

Passage par valeur:

```
void incrementer(int a)
{
      a = a + 1;
}

void main()
{
      int X;
      incrementer(X);
}
```





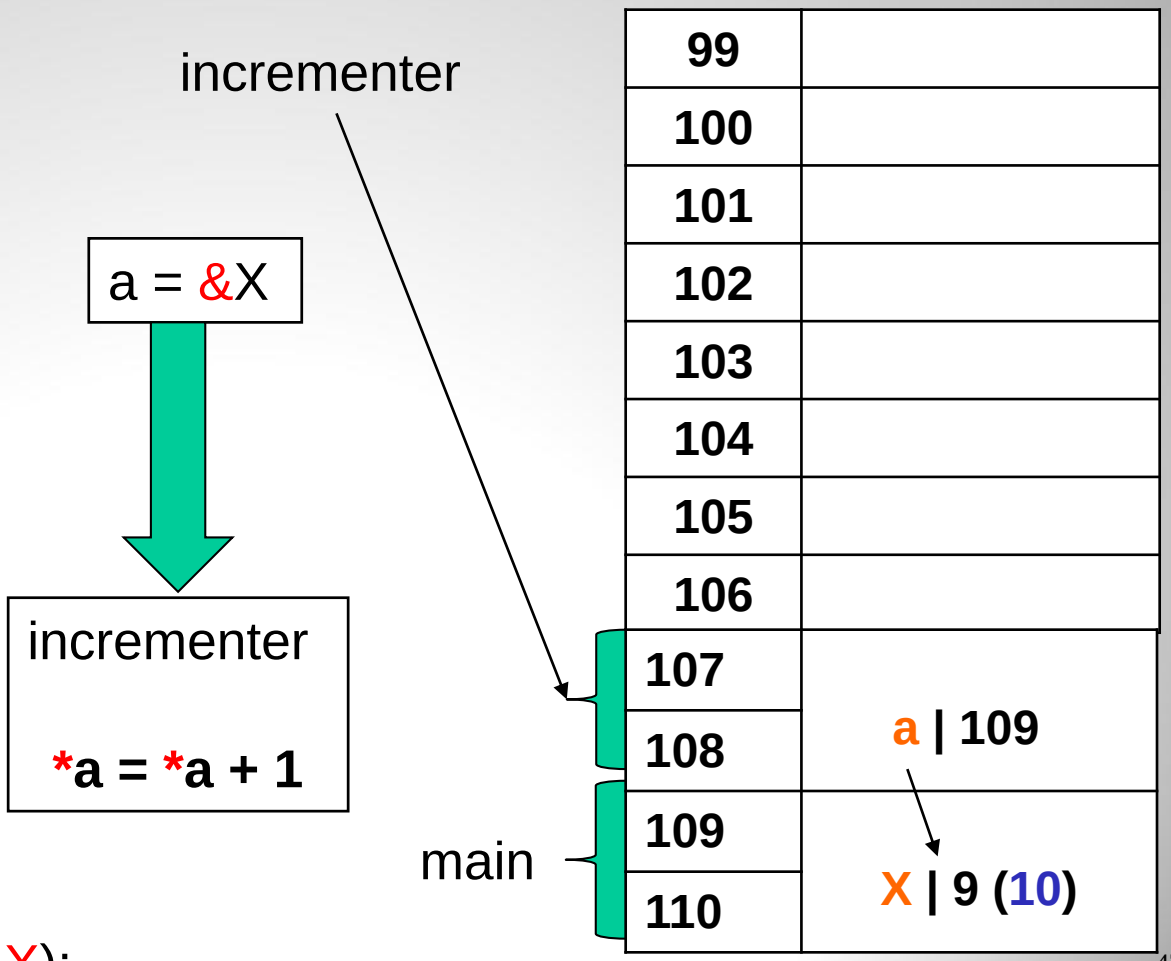
printf(‘’la valeur de X %d \n’’,X);

Les pointeurs et les fonction (passage par valeur et passage par adresse)

Exemple:

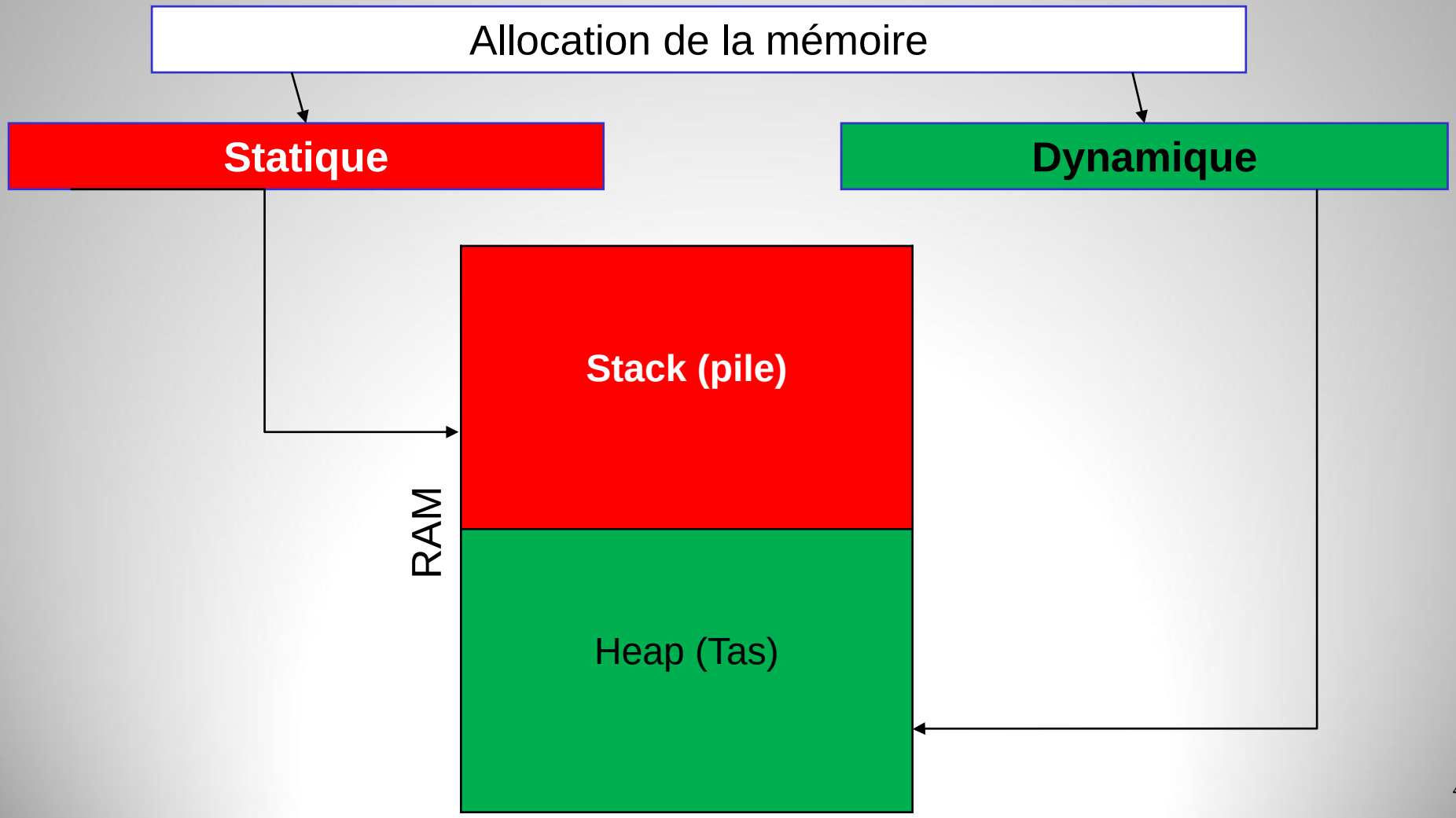
Passage par adresse:

```
void incrementer(int *a)
{
      *a = *a + 1;
}

void main()
{
      int X;
      incrementer(&X);
}
```



printf(‘’la valeur de X %d \n’’,X);

L’allocation de la mémoire



L'allocation statique de la mémoire

```
#include <stdio.h>

void main() {

float  salaires[7];

int i;

for (i = 0; i < 7 ; ++i) {

printf(" Donner le salaire de l'ouvrier %d ",i+1);
scanf("%f", &salaire[i]);

}
}
```

10000
15000
20000
13000
14000
18200
19000

L'allocation statique de la mémoire

```
#include <stdio.h>

void main() {

float  salaires[13];

int i;

for (i = 0; i < 13 ; ++i) {

printf(" Donner le salaire de l'ouvrier %d ",i+1);
scanf("%f", &salaire[i]);

}
}
```

10000
15000
20000
13000
14000
19000
15000
20000
31000
40000
18000
17000
19000

Les limites de L'allocation statique de la mémoire

```
#include <stdio.h>

void main() {

float  salaires[1000];
int N, i;

do{
    printf(" Donner le salaire de l'ouvrier %d ");
    scanf("%d",&N)
}while(N>1 || N<1000);

for (i = 0; i < N ; ++i) {

     printf(" Donner le salaire de l'ouvrier %d ",i+1);

    scanf("%f", &salaires[i]);
}
}
```

N	&
10000	100
15000	101
20000	102
13000	103
14000	104
☺	**106**
☺	.
☺	.
☺	.
☺	.
☺	.
☺	**1000**

Solution: L'allocation dynamique de la mémoire

<stdlib.h>

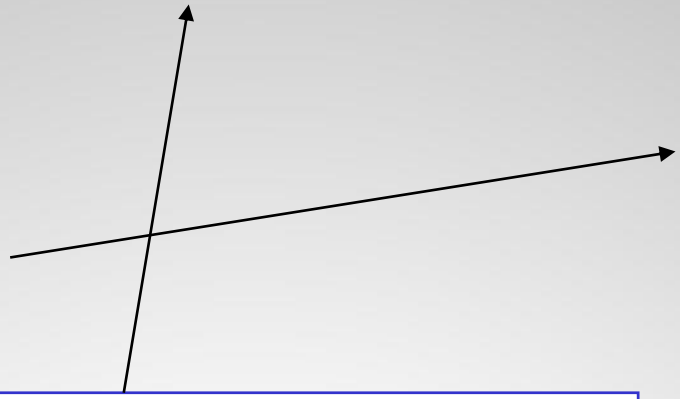
malloc() calloc() realloc() free()

- La fonction **malloc()** permet d'allouer dynamiquement un objet d'une taille donnée;
- La fonction **calloc()** fait de même en initialisant chaque multiplet à zéro;
- La fonction **realloc()** permet d'augmenter ou de diminuer la taille d'un bloc précédemment alloué avec **malloc()** ou **calloc()**;

malloc()

```
void* malloc(unsigned int);
```

```
void* malloc(6);
```



Le nombre d'octets que la fonction malloc va réserver aux nouveaux de la mémoire RAM

Heap

100	
101	
102	
103	
104	
105	
106	
107	
108	
109	
110	
111	
112	
113	
114	
115	

malloc()

```
void* malloc(unsigned int);
int *P;
P = malloc(6);
```

Stack

20	100
21	P
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	

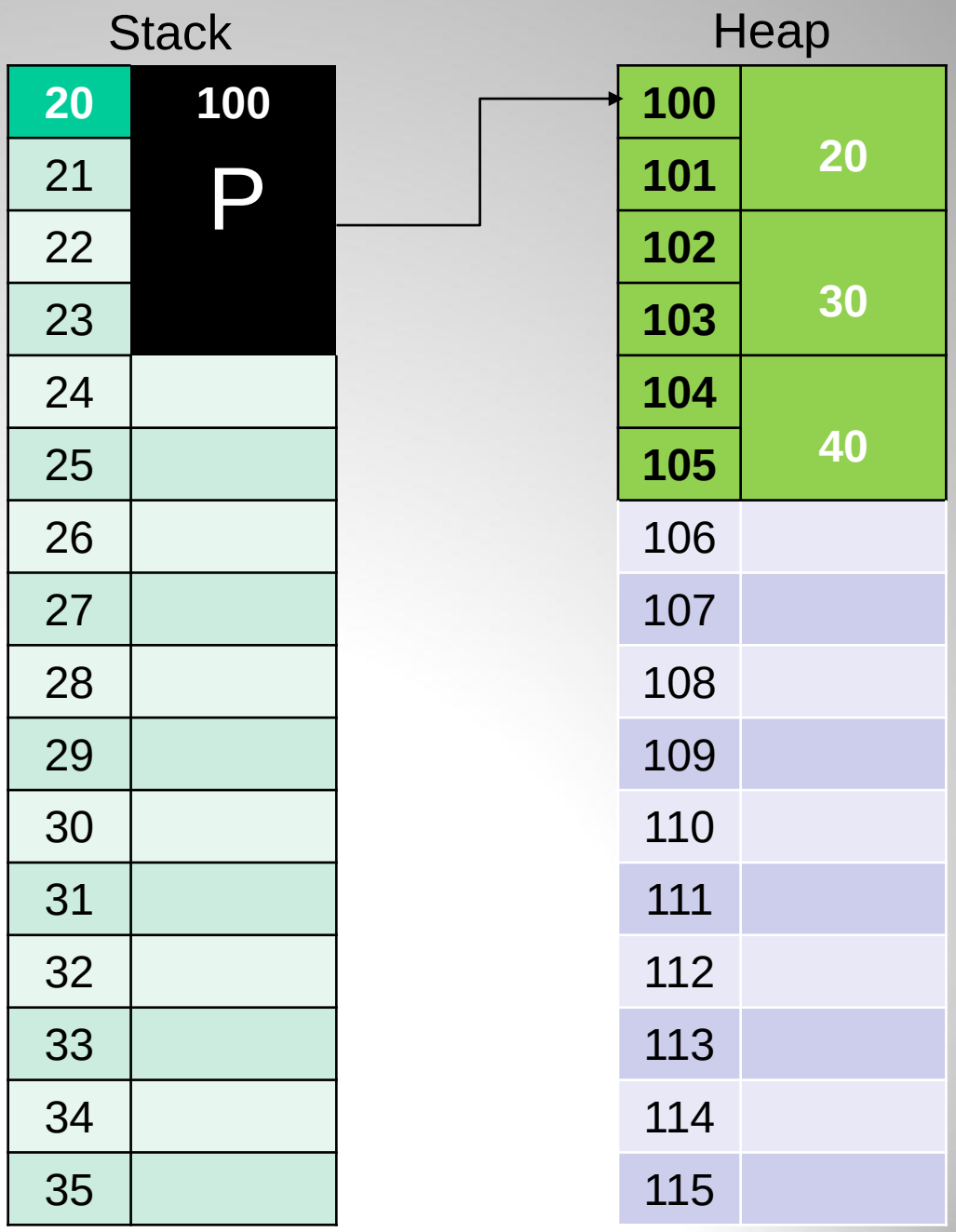
Heap

100	
101	
102	
103	
104	
105	
106	
107	
108	
109	
110	
111	
112	
113	
114	
115	######

malloc()

```
Void* malloc(unsigned int);
Int *P;
P = malloc(3*sizeof(int));
*P = 20;
*(P+1) = 30;
P[2] = 40;
```

Dépend du système: 1 **int = 2 octets** ou bien **1int = 4 octets.**

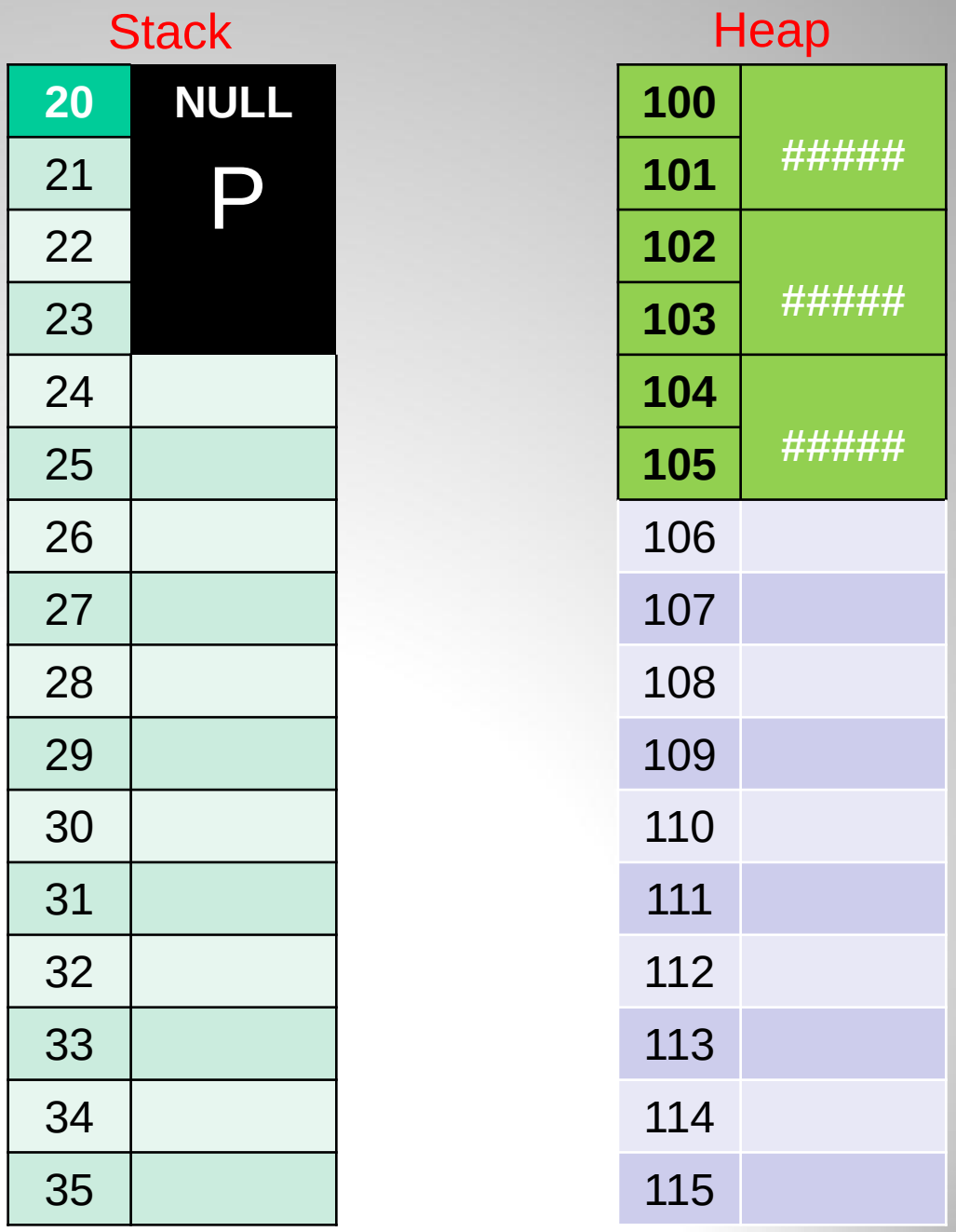


malloc()

Void* malloc(unsigned int);

Int *P;

P = malloc(3*sizeof(int));

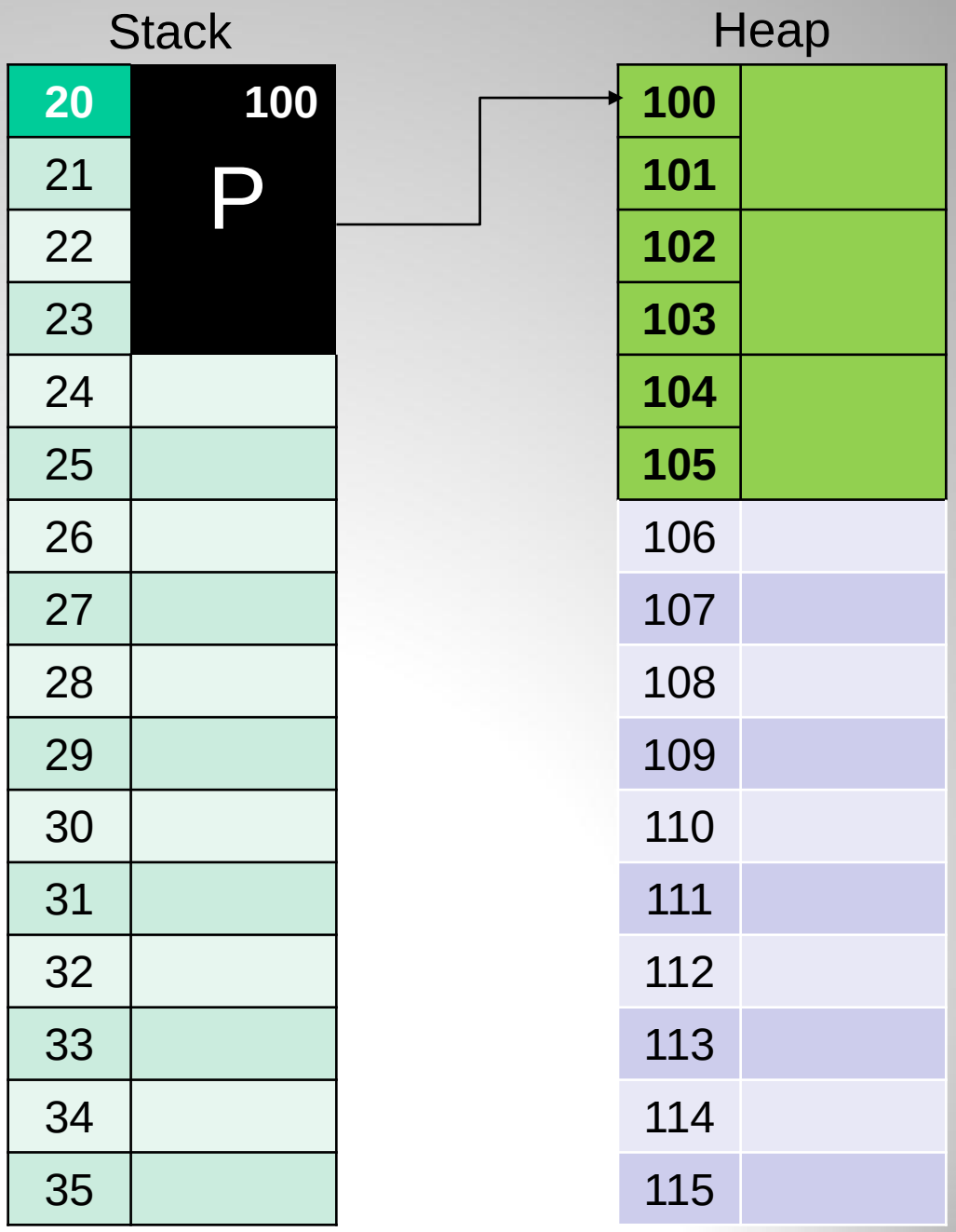


L'allocation dynamique de la mémoire: exemple

```
#include <stdio.h>
#include <stdlib.h>
void main() {

float*  salaires;
int N, i;

do{
    printf(" Donner le salaire de l'ouvrier %d ");
    scanf("%d",&N);
}while(N<0);
salaires = (float*)malloc(N*sizeof(float));
for (i = 0; i < N ; i++) {
     if (salaires == NULL)
          printf("Espace insuffisant");
     else
          printf(" Donner le salaire de l'ouvrier %d ",i+1);
          scanf("%f", &salaires[i]);
}}
```

free()

```
void* free(void*);

int *P;
P = malloc(3*sizeof(int));
.
.
.
free(P);
```



free()

```
void* free(void*);

int *P;
P = malloc(3*sizeof(int));
.
.
.
free(P);

P = NULL;
```

Stack

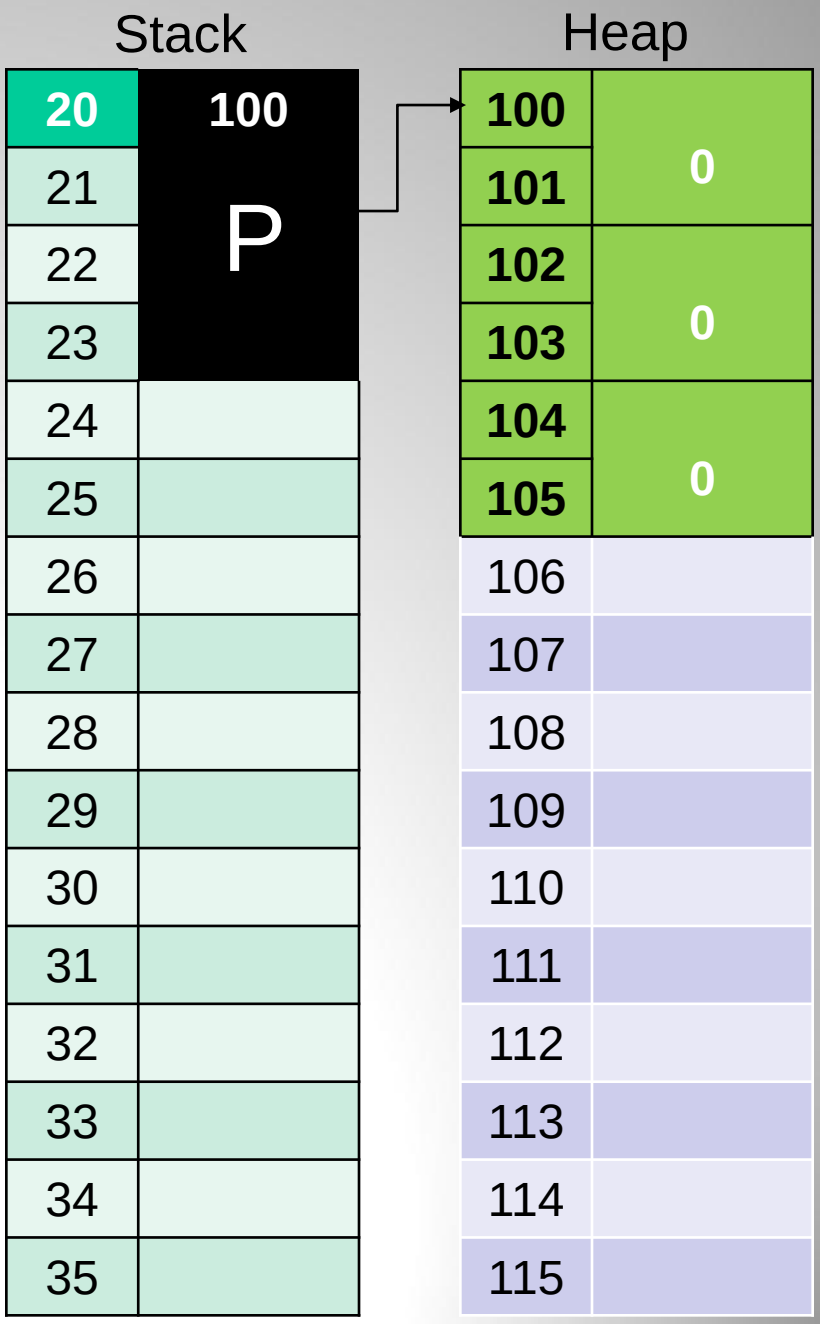
20	**NULL**
21	P
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	

Heap

100	**20**
101	
102	**30**
103	
104	**40**
105	
106	
107	
108	
109	
110	
111	
112	
113	
114	
115	

calloc()

```
Void* calloc(unsigned int, unsigned int);

Int *P;

P = calloc(3*sizeof(int));
```



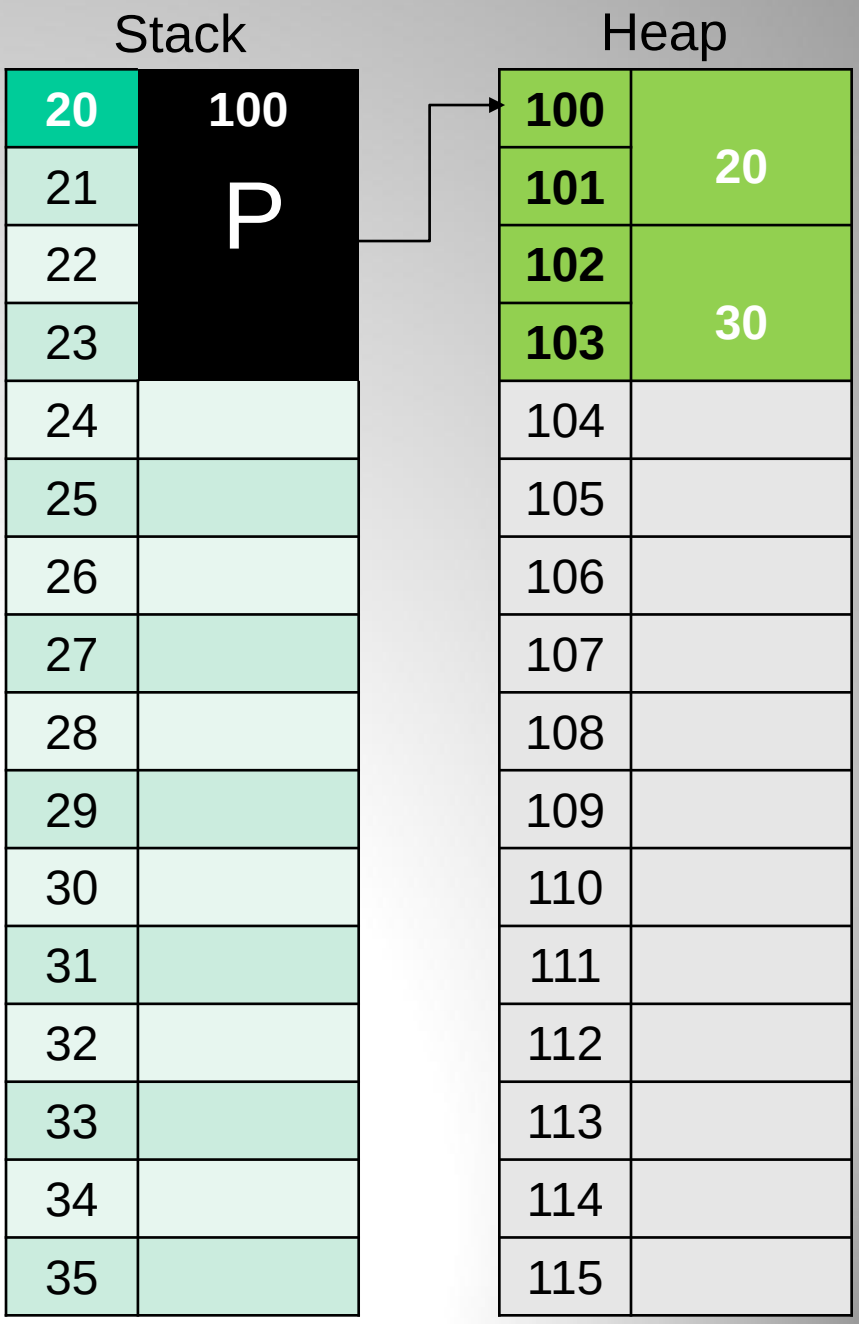
realloc()

Void* realloc(void*, unsigned int);

Int *P;

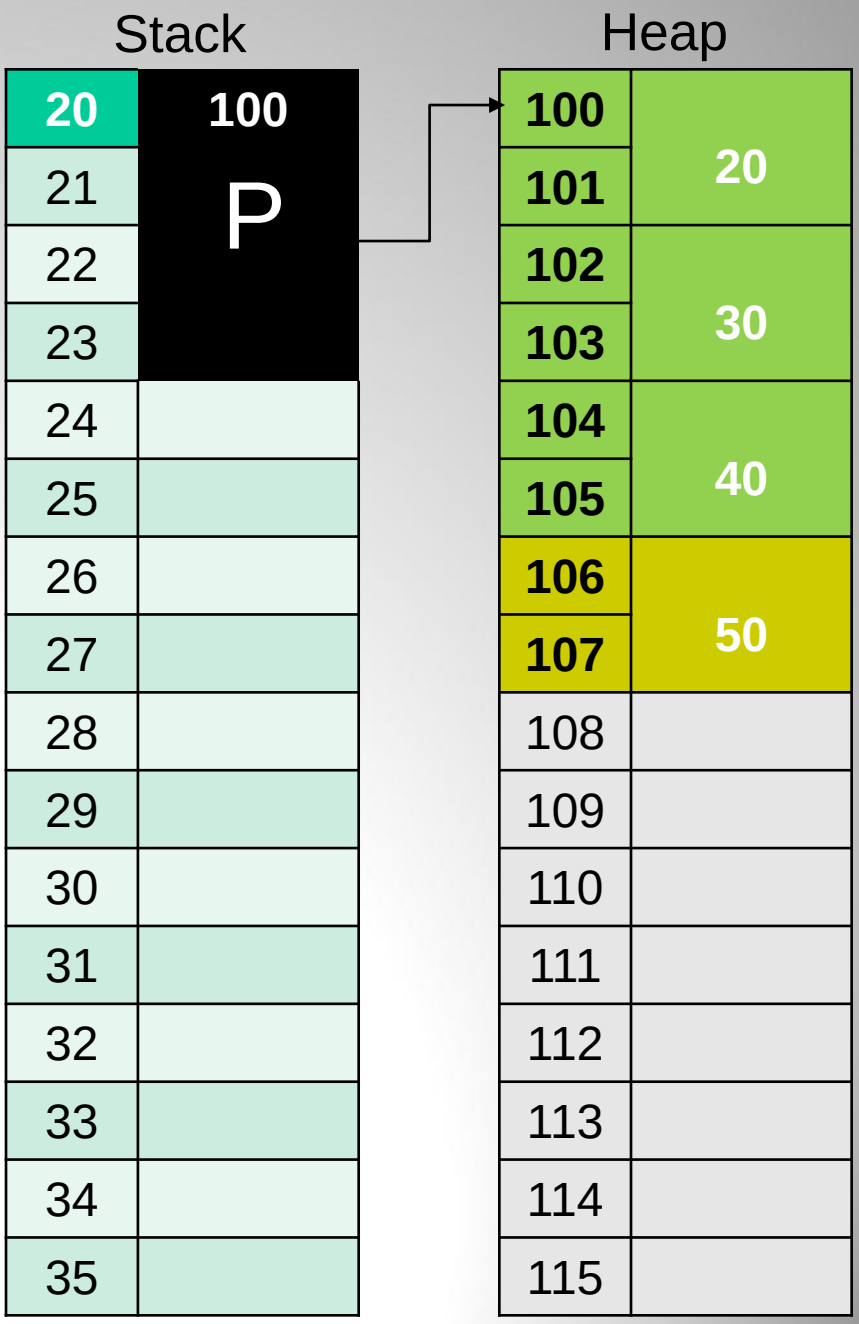
P = malloc(6);

P = realloc(P, 4);



realloc()

```
Void* realloc(void*, unsigned int);

Int *P;

P = malloc(6);

P = realloc(P, 8);

P[3] = 50;
```



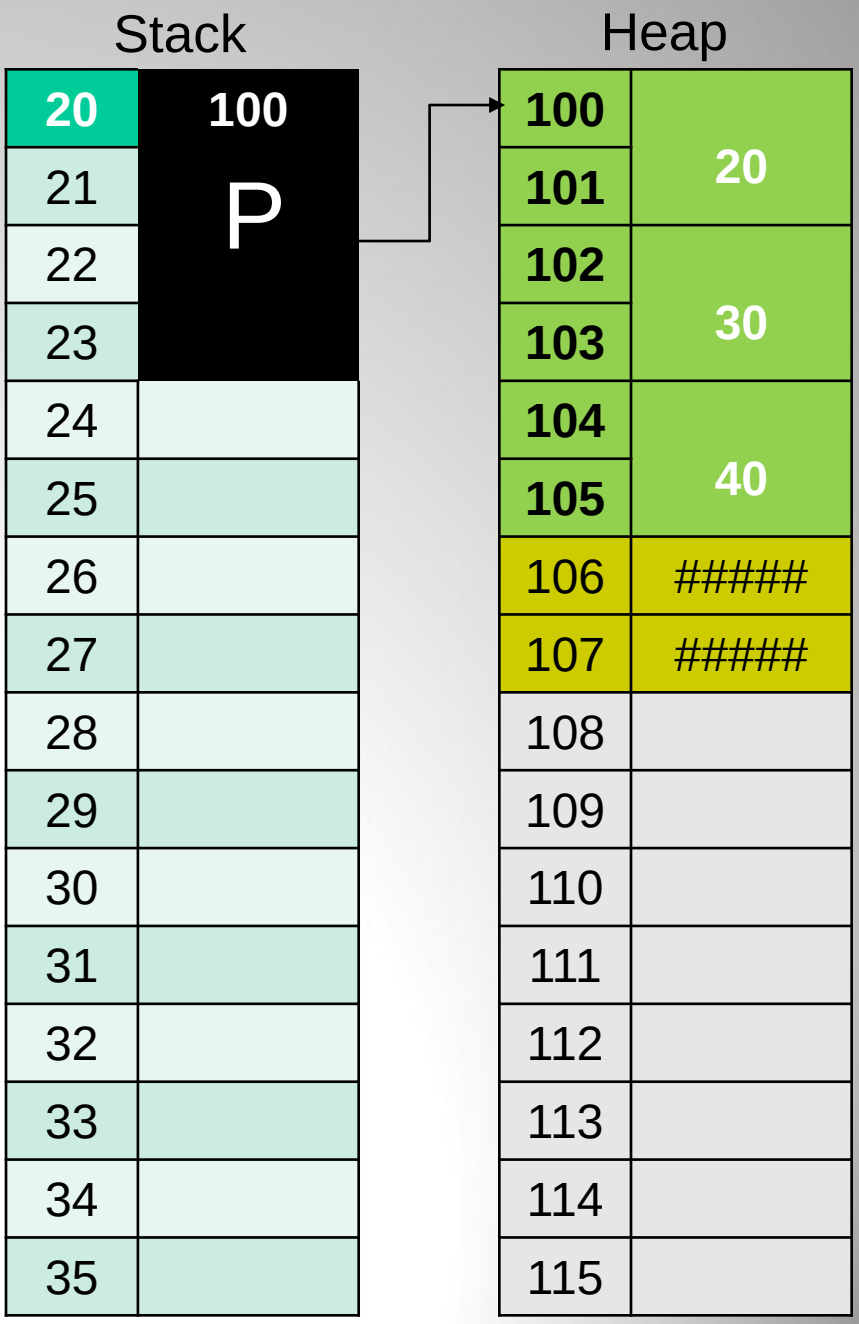
realloc()

Void* realloc(void*, unsigned int);

Int *P;

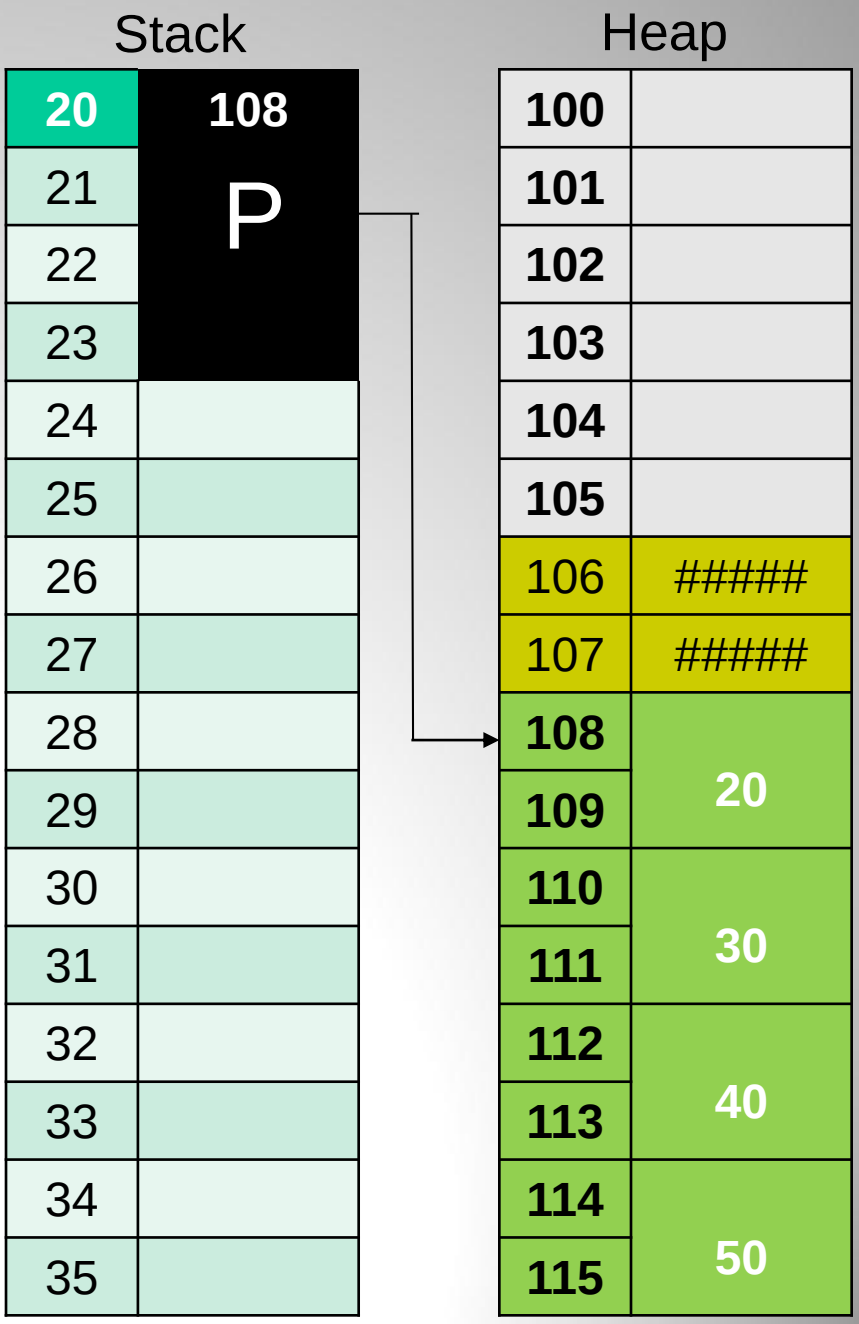
P = malloc(6);

.
.
.
.



realloc()

```
Void* realloc(void*, unsigned int);

Int *P;

P = malloc(6);

.
.
.
.
P = realloc(P, 8);

P[3] = 50;
```



Exemple: malloc

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

int main(void){
  int *tab;
  int i;

  tab=(int *)malloc(5*sizeof(int));
  if (tab == NULL) {
    printf("Mémoire non allouée.\n");
    exit(0);
  }
  else{
    printf("Mémoire allouée avec succès avec malloc \n");
    for(i=0 ; i < 5 ; i++){
      // *(tab+i) ou tab[i]
      *(tab+i)=i;
    }

    printf("Les éléments du tableau sont: ");
    for(i=0 ; i < 5 ; i++){

      // *(tab+i) ou tab[i]
      printf("%d, ", *(tab+i));
    }
  }
  return 0;
}
```

Exemple: calloc

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

int main(void){
  int *tab;
  int i;

  tab=(int*)calloc(5, sizeof(int));
  if (tab == NULL) {
    printf("Mémoire non allouée.\n");
    exit(0);
  }
  else{
    printf("Mémoire allouée avec succès avec malloc \n");
    for(i=0 ; i < 5 ; i++){
      // *(tab+i) ou tab[i]
      *(tab+i)=i;
    }

    printf("Les éléments du tableau sont: ");
    for(i=0 ; i < 5 ; i++){

      // *(tab+i) ou tab[i]
      printf("%d, ", *(tab+i));
    }
  }
  return 0;
}
```

Exemple: free

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

int main(void){
  int *tab;
  int i;

  tab=(int*)calloc(5, sizeof(int));
  if (tab == NULL) {
    printf("Mémoire non allouée.\n");
    exit(0);
  }
  else{
    printf("Mémoire allouée avec succès avec malloc
\n");
    for(i=0 ; i < 5 ; i++){
      // *(tab+i) ou tab[i]
      *(tab+i)=i;
    }
```

Exemple: free

```
    printf("Les éléments du tableau sont: ");
    for(i=0 ; i < 5 ; i++){

      // *(tab+i) ou tab[i]
      printf("%d, ", *(tab+i));
    }

    // liberer l'espace reservé par tab
    free(tab);
  }
  return 0;
}
```

Exemple: realloc

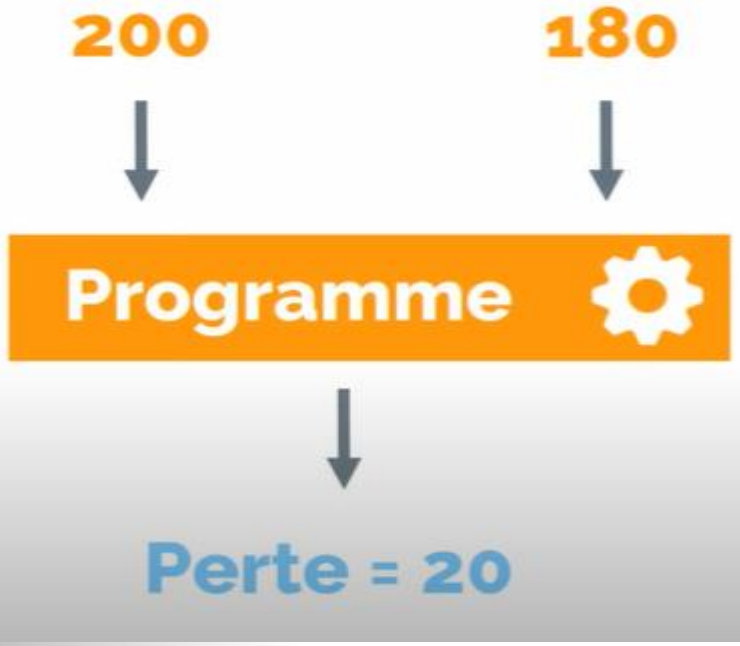
```
#include < stdio.h>
#include < stdlib.h>

int main(void){
  int *tab;
  int i, n;

  tab=(int*)calloc(5, sizeof(int));
  if (tab == NULL) {
    printf("Mémoire non allouée.\n");
    exit(0);
  }
  else{
    printf("Mémoire allouée avec succès avec malloc \n");
    for(i=0 ; i < 5 ; i++){
      // *(tab+i) ou tab[i]
      *(tab+i)=i;
    }

    printf("Les éléments du tableau sont: ");
    for(i=0 ; i < 5 ; i++){

      // *(tab+i) ou tab[i]
      printf("%d, ", *(tab+i));
    }
```

Exemple: realloc

```
// nouvelle taille
    n=10;
    tab=realloc(tab, n * sizeof(int));

    for(i=5 ; i < n ; i++){
      *(tab+i)=i;
    }

    printf("Les éléments du tableau après la “réallocation” sont : ");
    for(i=0 ; i < n ; i++){

      // *(tab+i) ou tab[i]
      printf("%d, ", *(tab+i));
    }

    // liberer l'espace reservé par tab
    free(tab);
  }
  return 0;
}
```

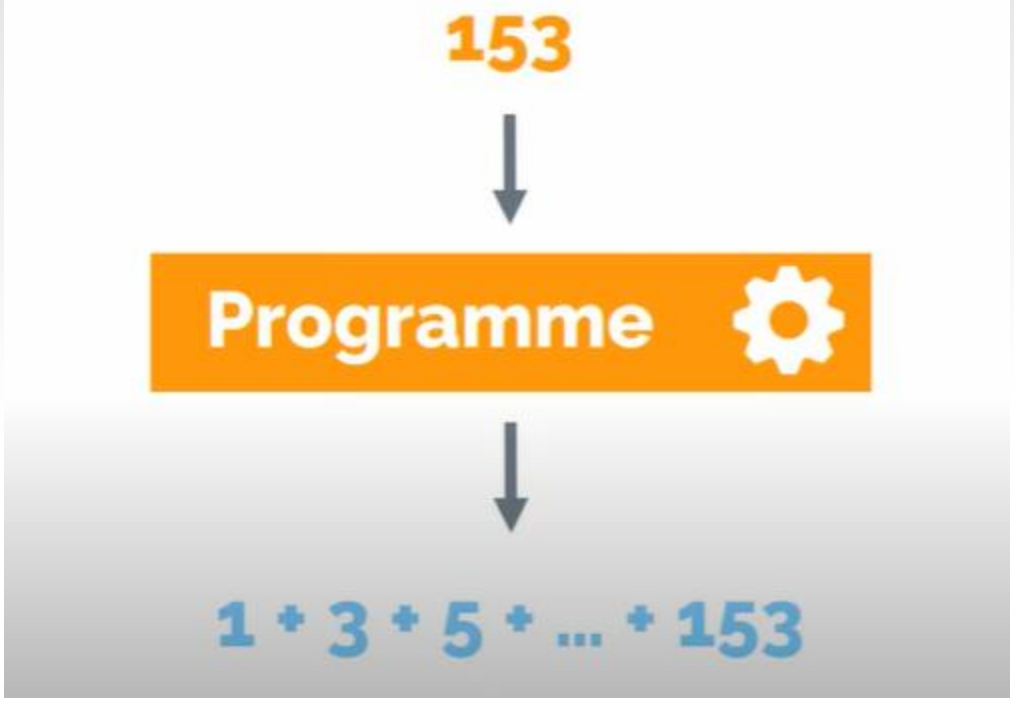
Exercice-1

À l'aide de l'allocation dynamique de mémoire, écrire un programme pour saisir le prix de revient et le prix de vente d'un produit, puis le programme vérifie si vous avez un profit ou une perte.



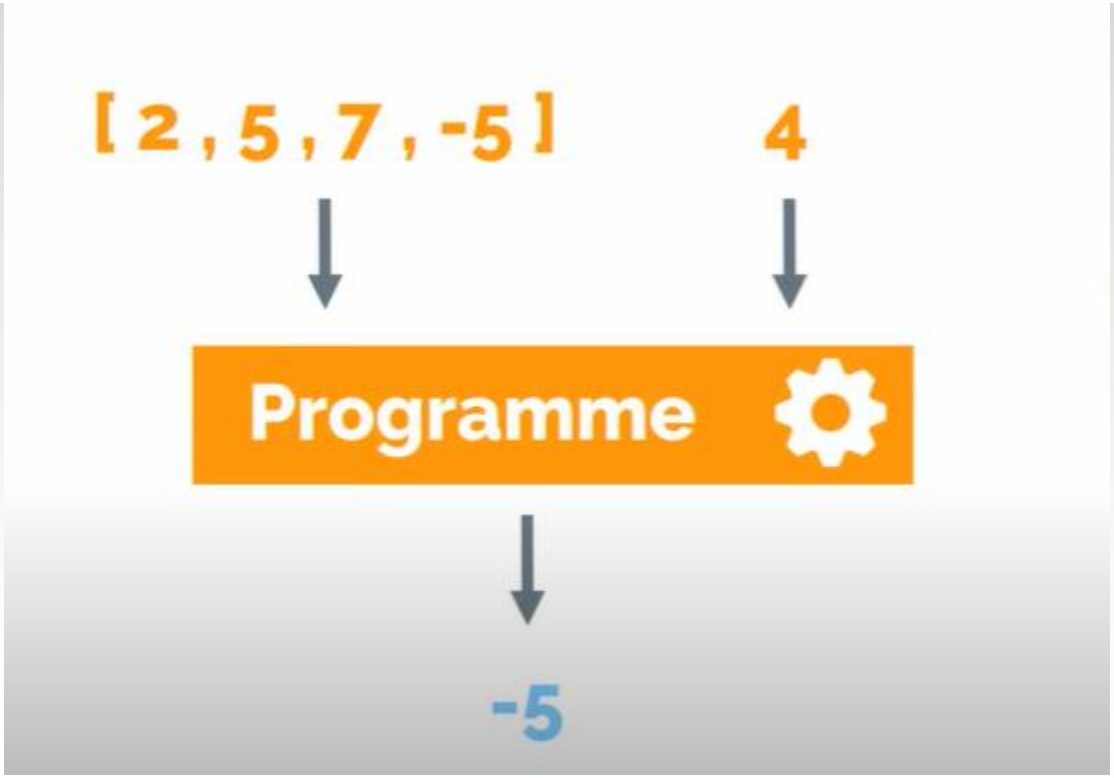
Exercice-2

À l'aide de l'allocation dynamique de mémoire, écrire un programme pour trouver la somme de tous les nombres impairs de 1 à n.



Exercice-3

Écrire un programme qui trouve le plus petit élément d'un tableau à l'aide de l'allocation dynamique de mémoire.



Exercice-4

Écrire un programme qui demande à l'utilisateur de remplir dynamiquement un tableau de n éléments, puis le programme calcule et affiche la moyenne des éléments du tableau. Le programme doit également demander à l'utilisateur s'il souhaite démarrer une autre opération ou quitter le programme.

NB: Penser à utiliser la fonction realloc() pour changer dynamiquement la taille du tableau.

Exercice-5-6-7

Écrivez un programme qui demande à l'utilisateur la taille d'un tableau, alloue dynamiquement de la mémoire pour ce tableau, demande à l'utilisateur d'entrer les éléments du tableau, calcule la somme de ces éléments, puis libère la mémoire allouée. Utilisez les fonctions **‘malloc’, ‘free’** et les boucles pour réaliser cela.

Écrivez un programme qui demande à l'utilisateur d'entrer une chaîne de caractères, alloue dynamiquement de la mémoire pour une copie de cette chaîne, copie la chaîne originale dans la copie, affiche la copie, puis libère la mémoire allouée. Utilisez les fonctions **‘malloc’,** free et les fonctions de manipulation de chaînes de caractères de la bibliothèque standard (**string.h**).

Écrivez un programme qui crée un tableau de taille initiale, puis demande à l'utilisateur d'entrer des éléments pour remplir ce tableau. Lorsque le tableau est plein, demandez à l'utilisateur s'il souhaite ajouter plus d'éléments. Si oui, utilisez la fonction **‘realloc’** pour augmenter la taille du tableau et continuez à remplir le tableau avec les nouveaux éléments. Une fois terminé, affichez le tableau final et libérez la mémoire allouée.

Les unions

Définition

Union est un type de données spécial disponible en C qui permet de stocker différents types de données, comme les structures, *mais dans le même emplacement mémoire.*

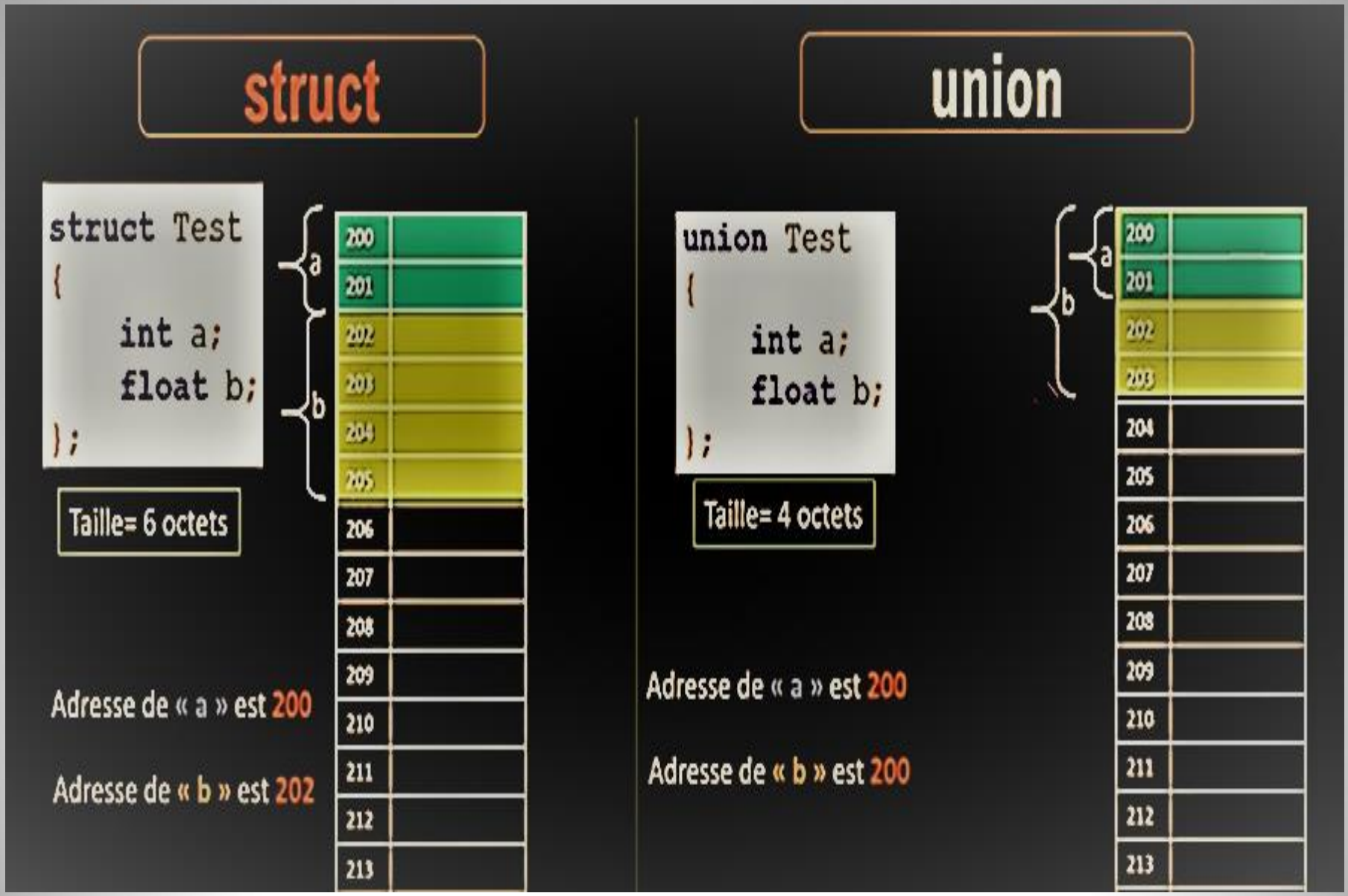
Vous pouvez définir une union avec plusieurs membres, mais un seul membre peut contenir *une valeur à la fois.*

Les Unions offrent un moyen efficace d'utiliser le même emplacement de mémoire pour des type différents

```
union [nom]
{
Type1   E1;
Type2   E2;

TypeN   EN;
};
```

```
union ABC
{
int     E1;
float   E2;
char    E3;
};
```

Les unions


struct
union
struct Test
{
int a;
float b;
};
union Test
{
int a;
float b;
};
a
b
200
201
202
203
204
205
206
207
208
209
210
211
212
213
Taille= 6 octets
Taille= 4 octets
Adresse de « a » est 200
Adresse de « b » est 202
Adresse de « a » est 200
Adresse de « b » est 200

Les unions: Exemple

```
union Test
        {
         char w;
         int x;
         float y;
         double z;

        } ;
```

char : 1 octet

int : 2 octets

float : 4 octets

double : 8 octets

La taille = 8 Octets

Les unions: Exemple

```
union Test
      {
       char w[15];
       int x;
       float y;
       double z;

      } ;
```

char : 1 octet

int : 2 octets

float : 4 octets

double : 8 octets

La taille = 15 Octets

Les unions

```
union Test
      {
       char a;
       int b;
      };
```

```
void main()
{
    union Test X;
    X.a='D';
    X.b=89;
    printf("La valeur de a =%c\n",X.a);
    printf("La valeur de b =%d\n",X.b);
}
```

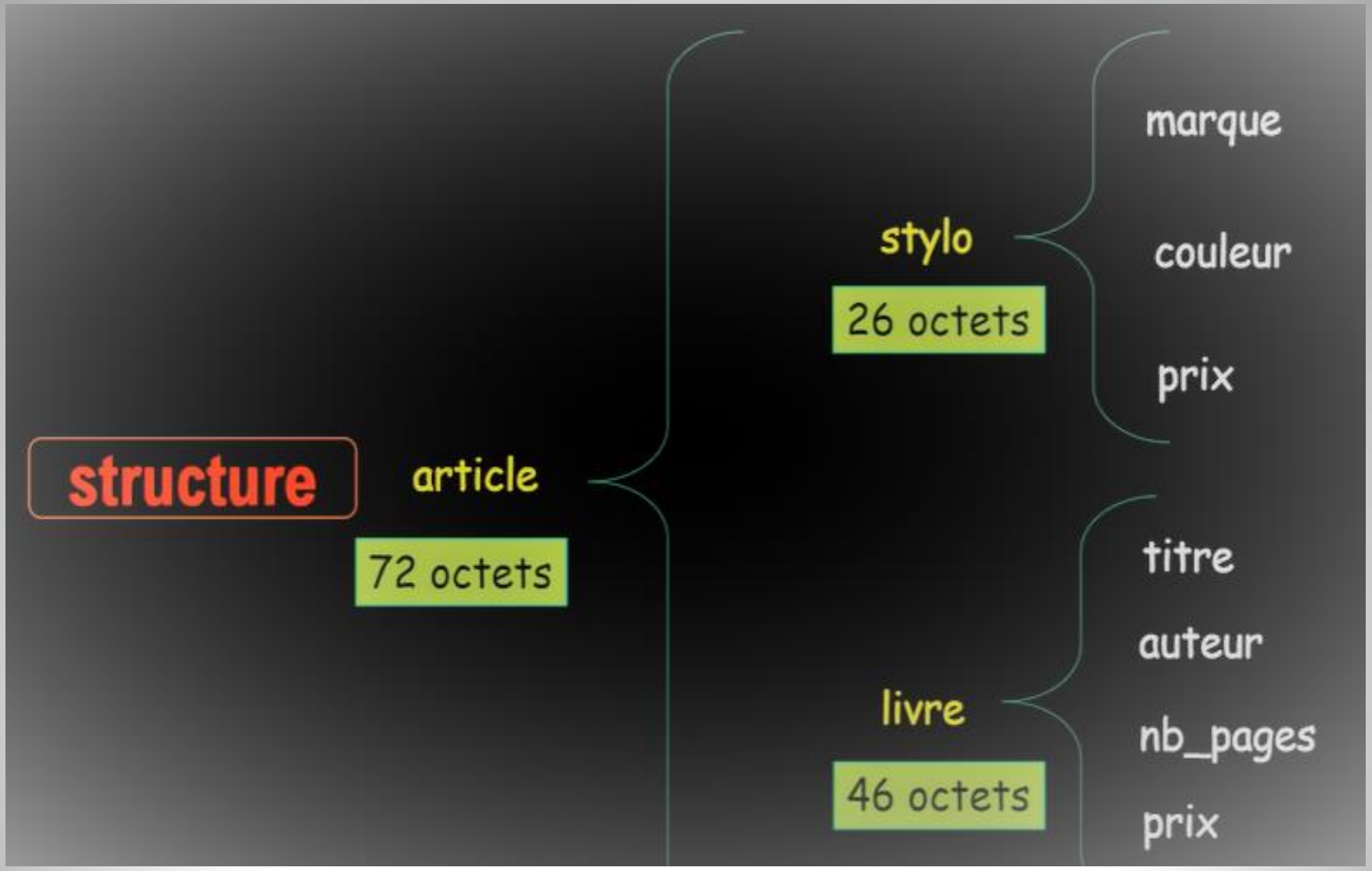
Les unions

```
union Test
      {
       char a;
       int b;
      };
```

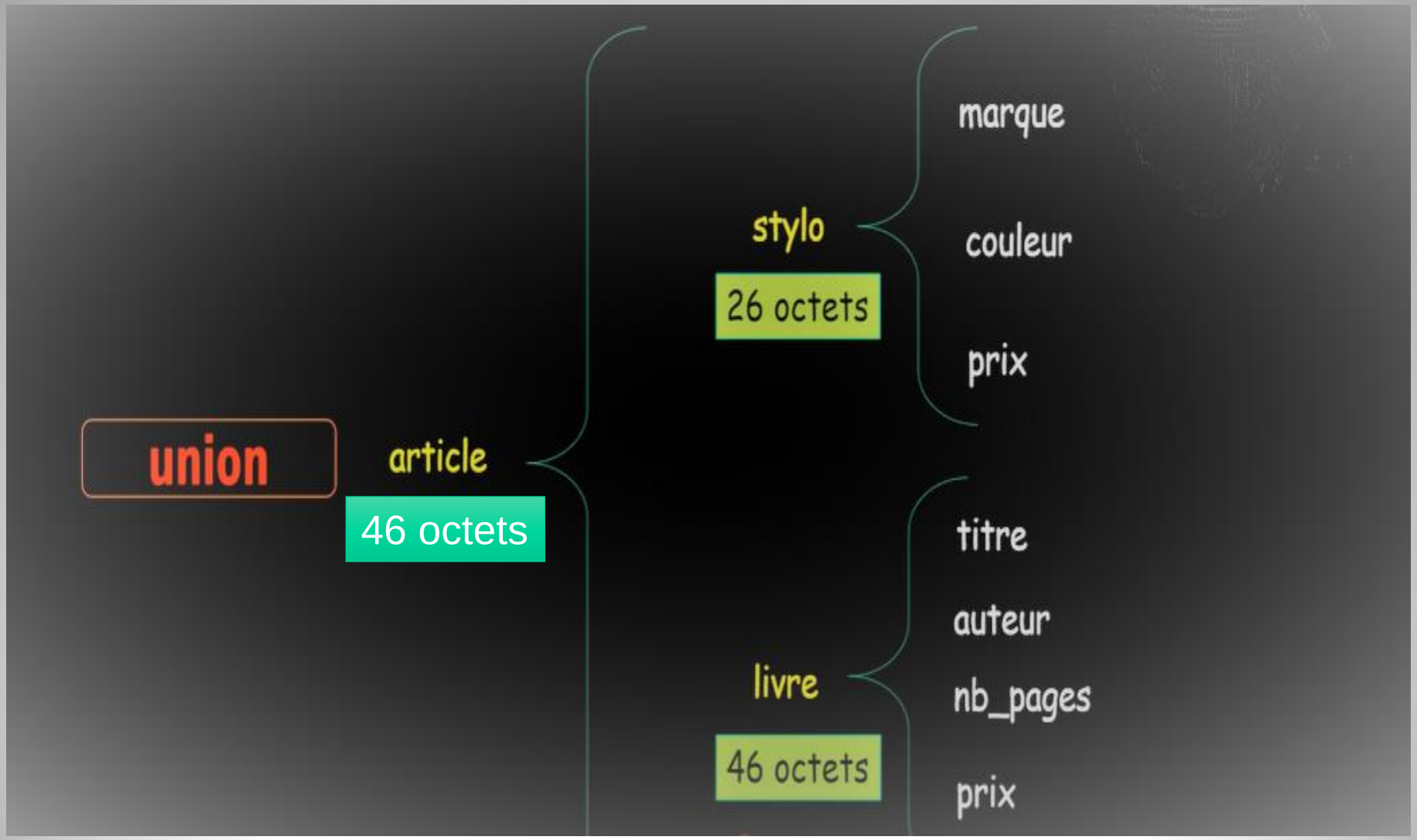
```
void main()
{
    union Test X;
    X.a='D';
    X.b=89;
    printf("La valeur de a =%c\n",X.a);
    printf("La valeur de b =%d\n",X.b);
}
```

Exemple d'utilisation

Exemple d'utilisation



Exemple d'utilisation



Exercices

Exercice 1 :

Écrire un programme qui permet de **gérer un carnet d'adresses**. Le programme doit permettre d'**ajouter** des contacts, de les **afficher** et de **rechercher** un contact par nom. Chaque contact est représenté par une structure contenant le **nom**, le **numéro de téléphone** et **l'adresse email**.

Étapes :

- ✓ Définir une **structure** Contact avec les champs appropriés.
- ✓ Utiliser un **tableau dynamique** pour stocker les contacts.
- ✓ Implémenter des **fonctions** pour ajouter un contact, afficher tous les contacts et rechercher un contact par nom.
- ✓ Utiliser des **boucles** pour permettre plusieurs opérations jusqu'à ce que l'utilisateur choisisse de quitter.

Exercices

Exercice 2 :

Écrire un programme pour **gérer une bibliothèque**. Le programme doit permettre d'**ajouter** des livres, de les **afficher**, de les **trier** par titre et de **rechercher** un livre par titre. Chaque livre est représenté par une structure contenant le titre, l'auteur et l'année de publication.

Étapes :

- ✓ Définir une **structure** Livre avec les champs appropriés.
- ✓ Utiliser un **tableau dynamique** pour stocker les livres.
- ✓ Implémenter des **fonctions** pour ajouter un livre, afficher tous les livres, trier les livres par titre et rechercher un livre par titre.
- ✓ Utiliser des **boucles** pour permettre plusieurs opérations jusqu'à ce que l'utilisateur choisisse de quitter.

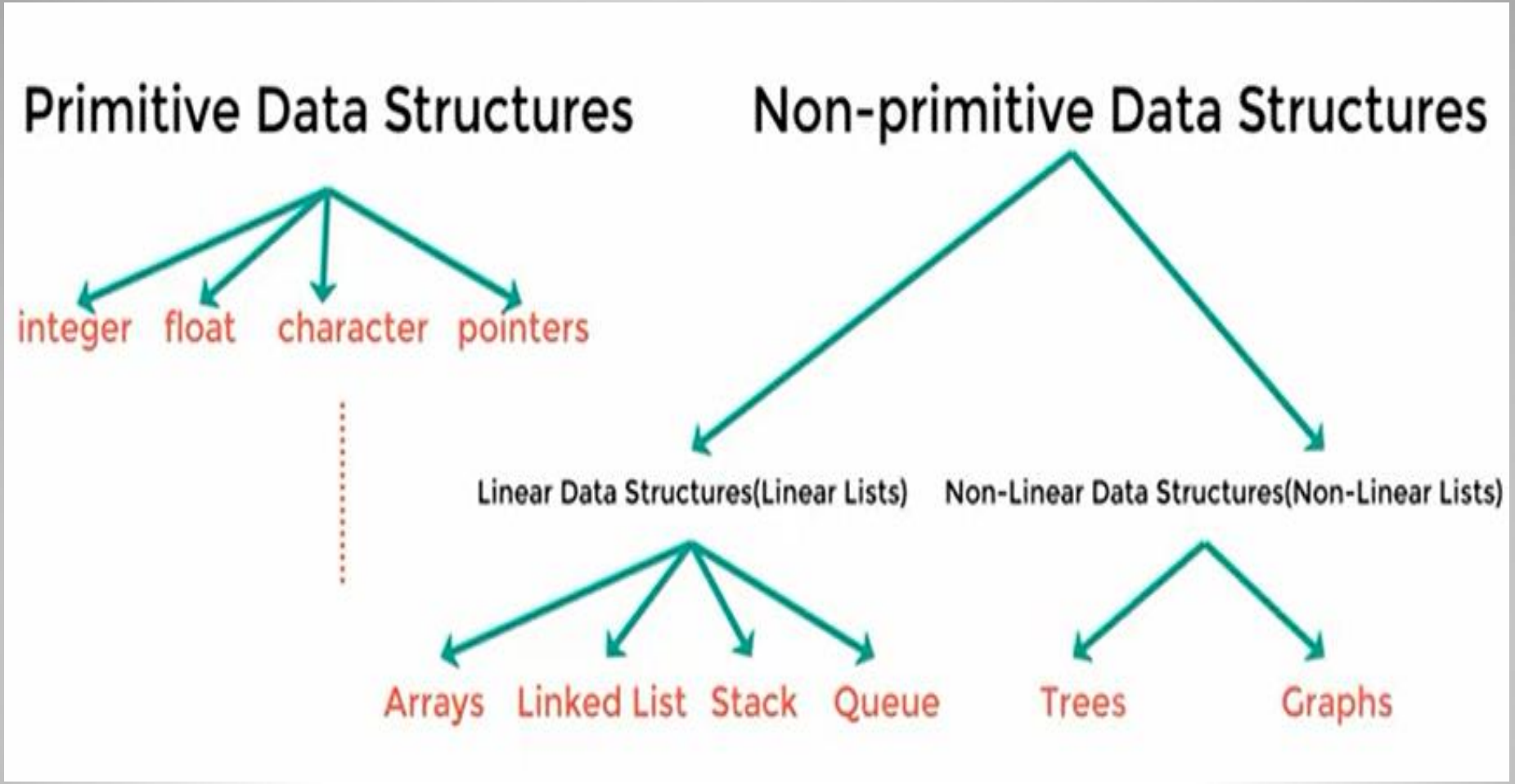
Data Structure

Data Structure is the way of storing and access data from memory so that data can be used efficiently.

Actually in our programming data stored in main memory(RAM) and to develop efficient software or firmware we need to care about memory.

To efficiently manage we required data structure.

Types of Data Structure



Types of Data Structure

Linear data structure:

A data structure is said to be linear if items are arranged in linear or sequential format.

Array Data Structure

Linear data structure:

- Array
- Stack
- Queue
- Linked List

Search

Types of Data Structure

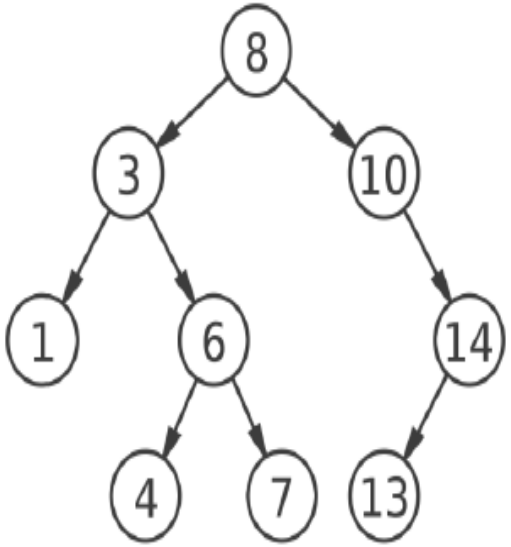
best case	Omega Notation, Ω
average case	theta notation, Θ
worst case	Big O notation, O

Types of Data Structure

Non-linear data structure:

Non-linear data structure does not support sequential format.

Tree data structure



Non-linear data structure:

- Tree
- Graph

*, /, %, ^, +, -

++, --

+=, -=, /=, *=

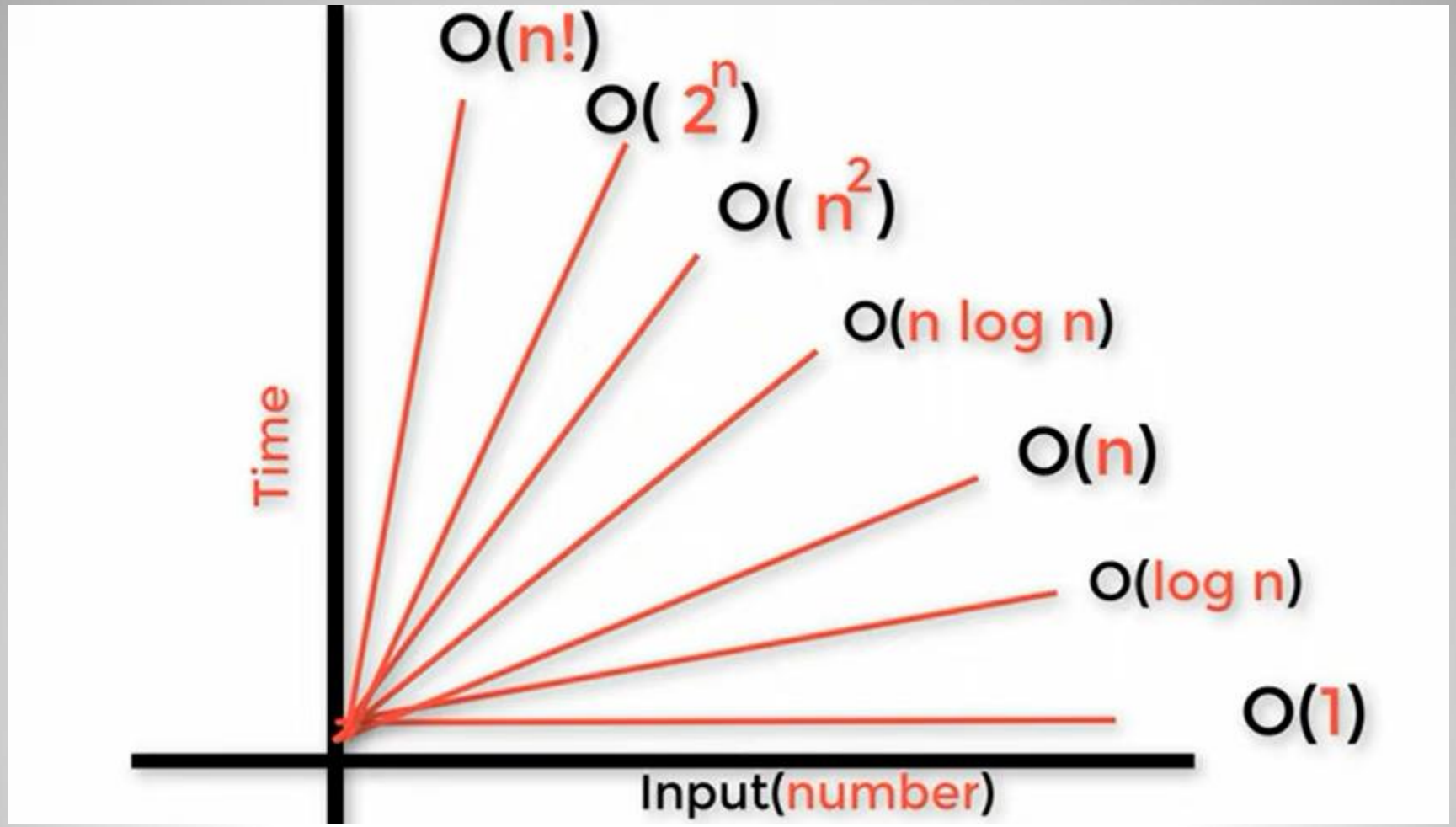
if, else, ifelse

constant time

Complexity

```
sum;
for(i = 1;i<=n;i++)//n
sum=sum+i; //constant time 1
```

Time Complexity : 1 + n = O(n)


O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n)
O(1)
Time
Input(number)

Stack Data Structure

- It is type of **linear data structure**.
- It follows **LIFO** (Last In First Out) property.
- It has only one pointer **TOP** that points the last or top most element of Stack.
- Insertion and Deletion in stack can only be done from top only.
- Insertion in stack is also known as a **PUSH operation**.
- Deletion from stack is also known as **POP operation** in stack.

Definition

"Stack is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Stack implementation

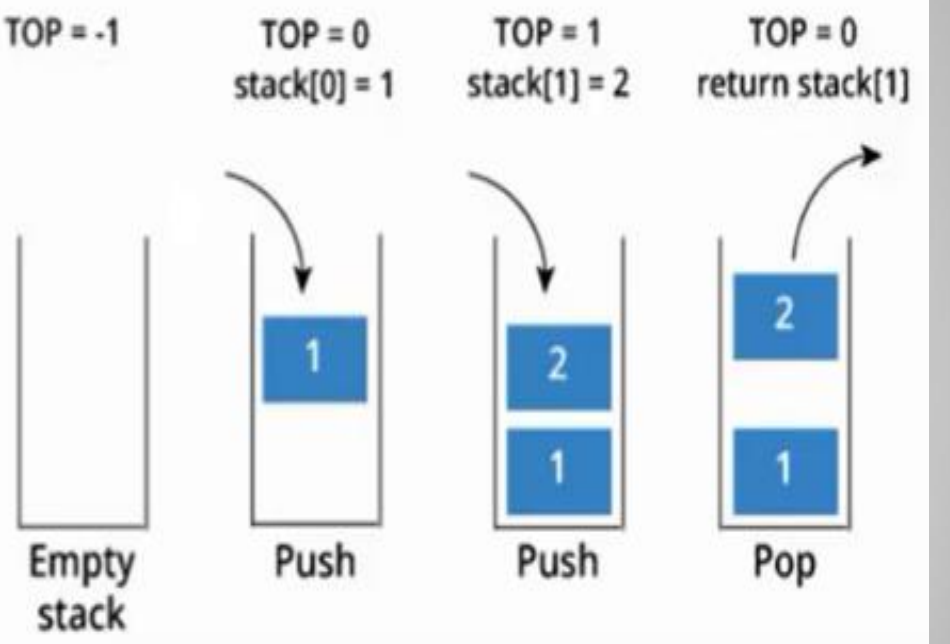
- ✓ Stack implementation using array
- ✓ Stack implementation using linked list

Stack Data Structure

How stack works

The operations work as follows:

1) A variable called TOP is used to keep track of the top element in the stack.
2) When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3) On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4) On popping an element, we return the element pointed to by TOP and reduce its value.
5) Before pushing, we check if stack is already full
6) Before popping, we check if stack is already empty

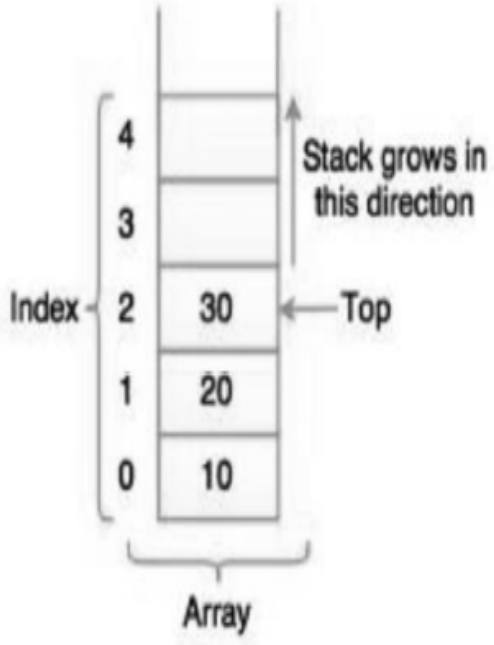


Stack using Array

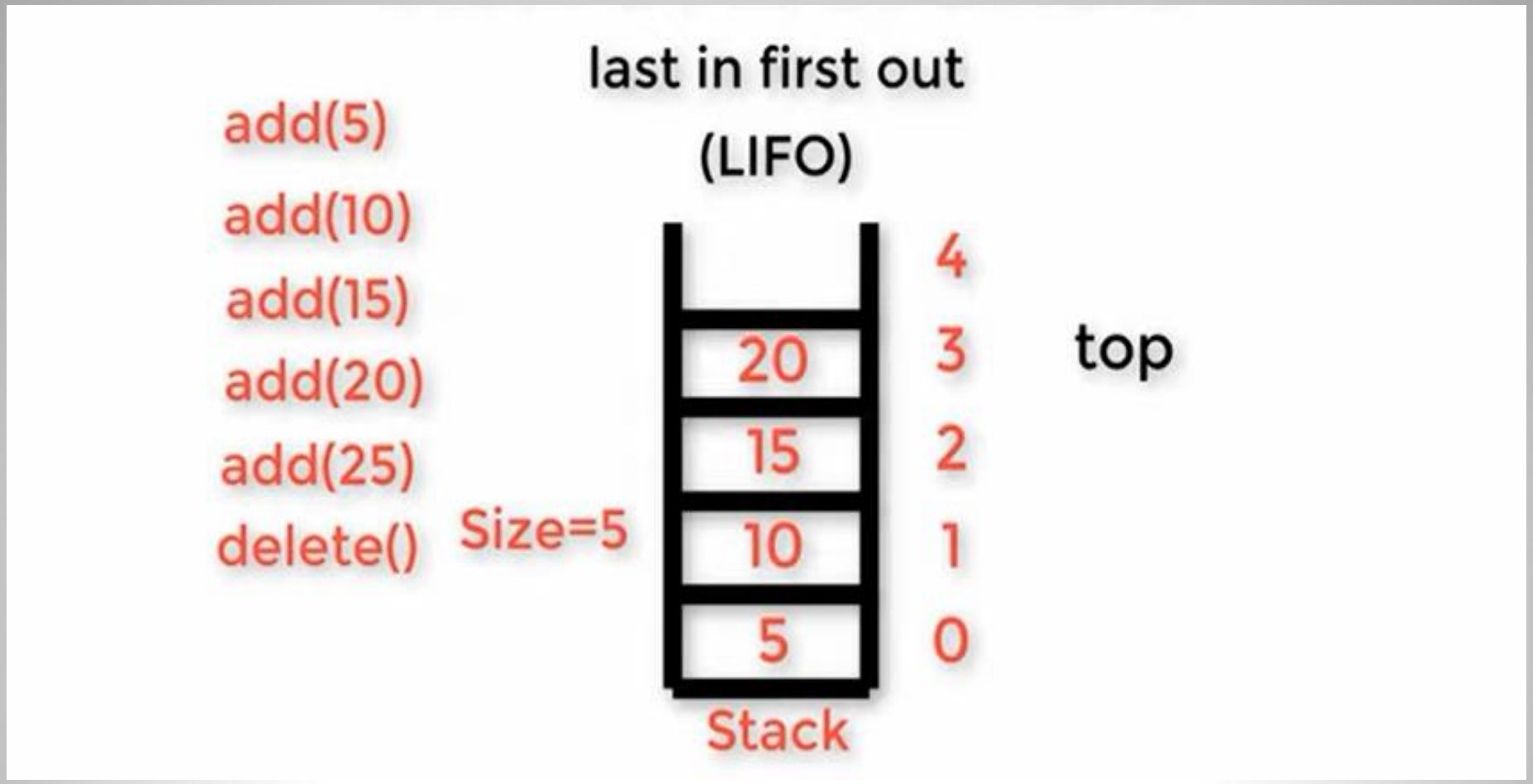
- Stack can be implemented using one-dimensional array.
- One-dimensional array is used to hold elements of a stack.
- Implementing a stack using array can store fixed number of data values.
- In a stack, initially top is set to -1.
- Top is used to keep track of the index of the top most element.

Following table shows the Position of Top which indicates the status of stack:

Position of Top	Status of Stack
-1	Stack is empty. (Underflow)
0	Only one element in a stack.
N - 1	Stack is full.
N	Stack is overflow. (Overflow state)



Stack Data Structure



Stack Data Structure

last in first out

(LIFO)

Array-Based Implementation of the ADT Stack

push(val) pop() getTop() isEmpty()

Stack Data Structure

last in first out

(LIFO)

Array-Based Implementation of the ADT Stack

- `Push`: Add element to top of stack
- `Pop`: Remove element from top of stack
- `IsEmpty`: Check if stack is empty
- `IsFull`: Check if stack is full
- `Peek`: Get the value of the top element without removing it

Use of stack

The most common uses of a stack are:

✓ To reverse a word

Put all the letters in a stack and pop them out.

Because of **LIFO** order of stack, you will get the letters in reverse order.

✓ In compilers

Compilers use stack to calculate the value of expressions like 2+4/5*(7-9).

✓ In browsers

The back button in a browser sa0ves all the **urls** you have visited previously in a stack. Each time you visit a new

page, it is added on top of the stack. When you press the back button, the current URL is removed from the

stack and the previous **url** is accessed.

Stack declaration

```
#define MAX_SIZE 10
#define DATA_TYPE char

typedef struct{
    DATA_TYPE elements[MAX_SIZE];
    int top;
}Stack;
```

- Stack function prototypes

```
void Init(Stack *);
int IsFull(Stack *);
int IsEmpty(Stack *);
int Push(DATA_TYPE, Stack *);
int Pop(DATA_TYPE *, Stack *);
int StackSize(Stack *);
void ClearStack(Stack *);
```

Stack Implementation

- Initialization of stack.

TOP points to the top-most element of stack.

1) TOP: = -1;

2) Exit

```
void Init(Stack *ptr_stack){

    ptr_stack->top = -1;
}
```

Stack Implementation

Stack is full.

1) IF TOP = MAX - 1 then

   return:=1;

1) Otherwise

   return:=0;

1) End of IF

2) Exit

```
int IsFull(Stack *ptr_stack){

    if(ptr_stack->top == MAX_SIZE - 1){
        return 1;
    } else {
        return 0;
    }
}
```

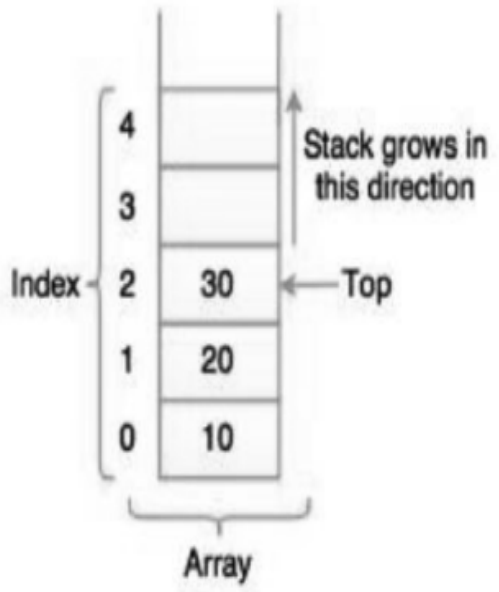


Fig. Implementation Stack using Array

Stack Implementation

Stack is empty.

1) IF TOP = - 1 then

   return:=1;

2) Otherwise

   return:=0;

3) End of IF

4) Exit

```
int IsEmpty(Stack *ptr_stack){

    if(ptr_stack->top == -1){
        return 1;
    } else {
        return 0;
    }
}
```

Stack Implementation

Push an item into stack.

1) IF TOP = MAX - 1 then return 0;

   Exit;

2) Otherwise

   TOP: = TOP + 1;

   STACK (TOP):= ITEM;

3) End of IF

1) Exit

```
int Push(DATA_TYPE element, Stack *ptr_stack){

    if(ptr_stack->top == MAX_SIZE - 1){ // Is stack full?
        return 0;
    } else {
        ptr_stack->top++;
        ptr_stack->elements[ptr_stack->top] = element;
        return 1;
    }
}
```

Stack Implementation

Pop an element from stack.

1) IF TOP = - 1 then return 0;

   Exit;

2) Otherwise

   ITEM: =STACK (TOP);

   TOP: = TOP - 1;

3) End of IF

1) Exit

```
int Pop(DATA_TYPE *ptr_element, Stack *ptr_stack){

    if(ptr_stack->top == -1){ // Is stack empty?
        return 0;
    } else {
        *ptr_element = ptr_stack->elements[ptr_stack->top];
        ptr_stack->top--;
        return 1;
    }
}
```

Stack Implementation

```
int StackSize(Stack *ptr_stack){

    if(ptr_stack->top == -1){ // Is stack empty?
        return 0;
    } else {
        return (ptr_stack->top + 1);
    }
}
```

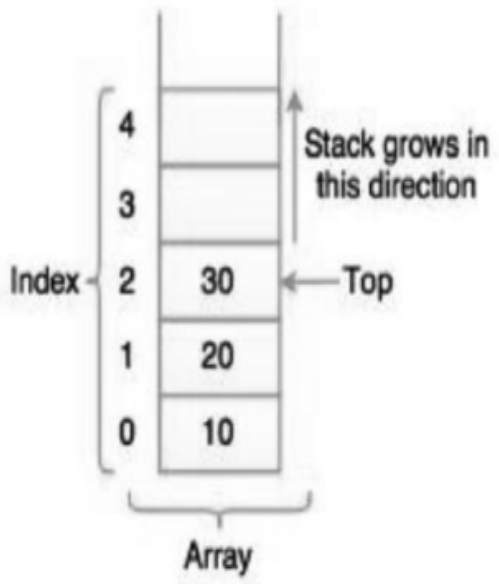


Fig. Implementation Stack using Array

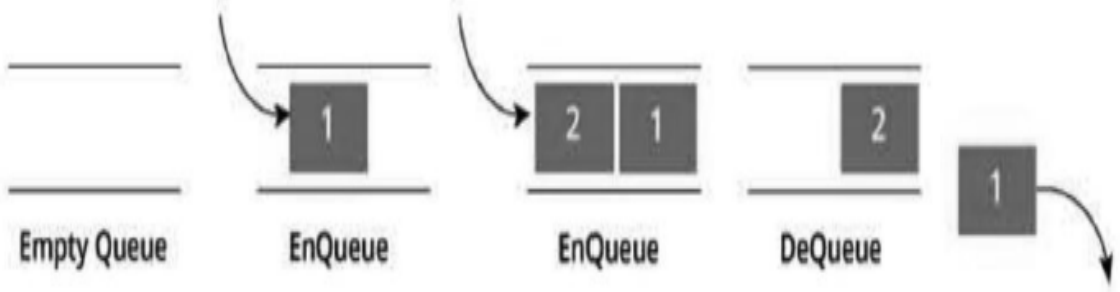
Queue

- ✓ Queue
- ✓ Front
- ✓ Rear
- ✓ Enqueue
- ✓ Dequeue
- ✓ First-In-First-Out (FIFO)

- Enqueue: inserts an element into the back of the queue.
- Dequeue: removes an element from the front of the queue.

Queue

- A queue is a collection of data that are added and removed based on the first-in-first-out (FIFO) principle.
- It means that the first element added to the queue will be the first one to be removed from the queue.
- Front points to the beginning of the queue and Rear points to the end of the queue.



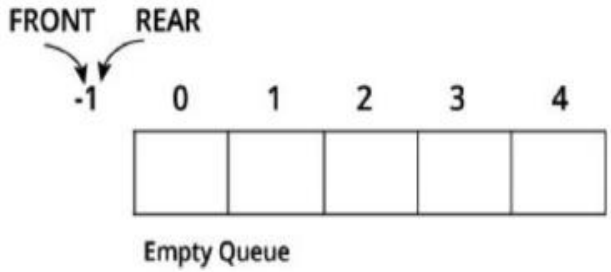
Queue: Linear Queue

How Queue Works

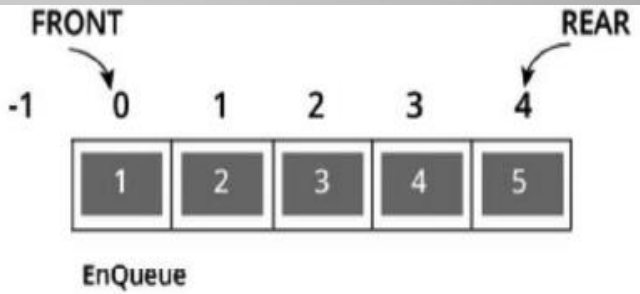
Queue operations work as follows:

1. Two pointers called `FRONT` and `REAR` are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of `FRONT` and `REAR` to -1.
3. On enqueing an element, we increase the value of `REAR` index and place the new element in the position pointed to by `REAR`.
4. On dequeueing an element, we return the value pointed to by `FRONT` and increase the `FRONT` index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of `FRONT` to 0.
8. When dequeing the last element, we reset the values of `FRONT` and `REAR` to -1.

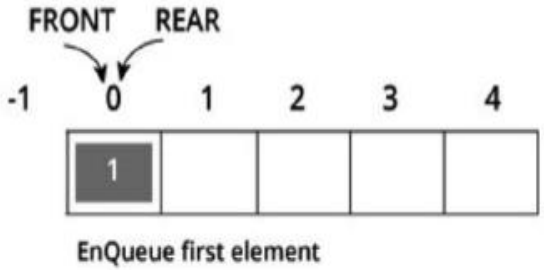
Queue



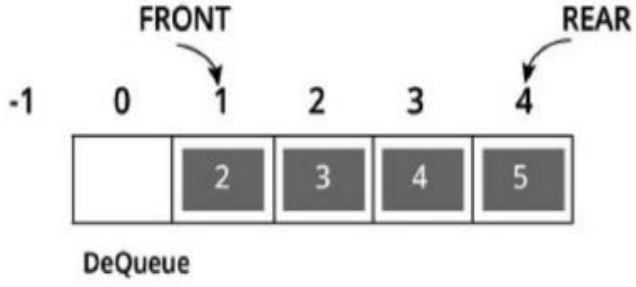
Empty Queue



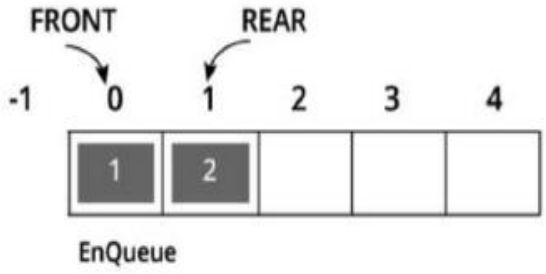
EnQueue



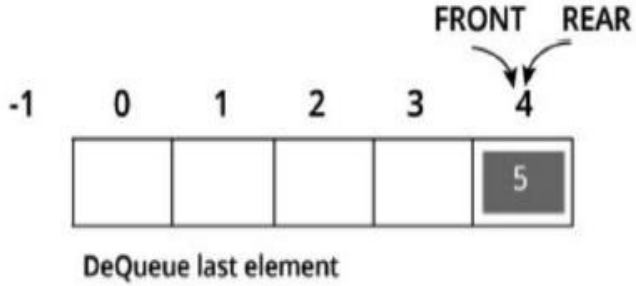
EnQueue first element



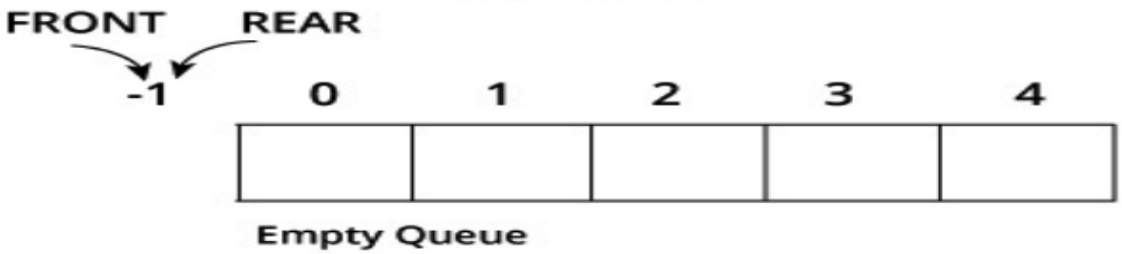
DeQueue



EnQueue



DeQueue last element

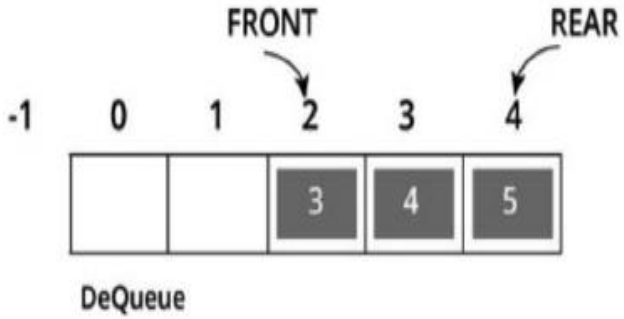


Empty Queue

Queue: Linear Queue

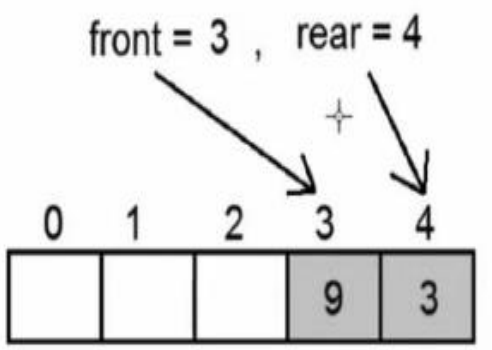
Drawback of Linear Queue

The linear queue suffers from serious drawback that performing some operations, we can not insert items into queue, even if there is space in the queue. Suppose we have queue of 5 elements and we insert 5 items into queue, and then delete some items, then queue has space, but at that condition we can not insert items into queue.

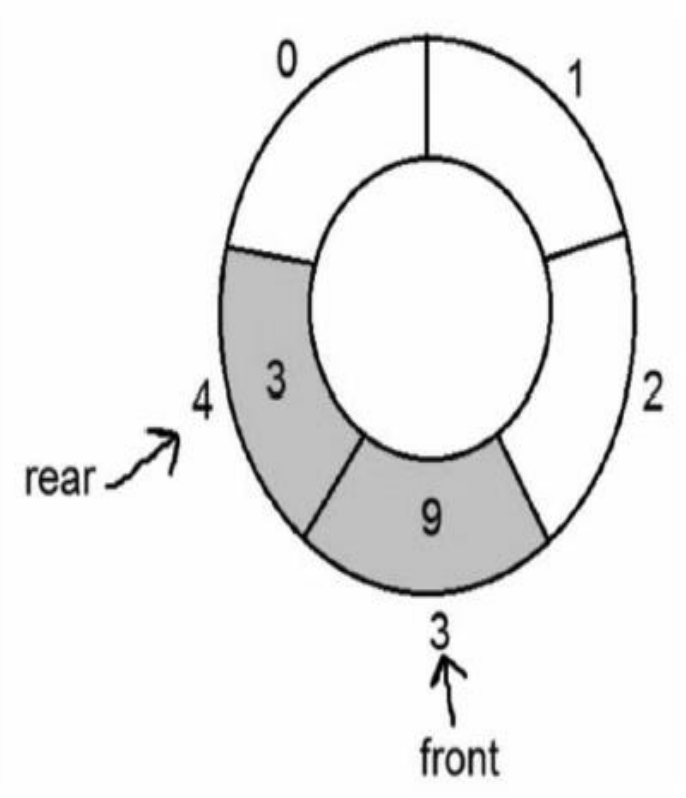


The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

Queue: Circular Queue



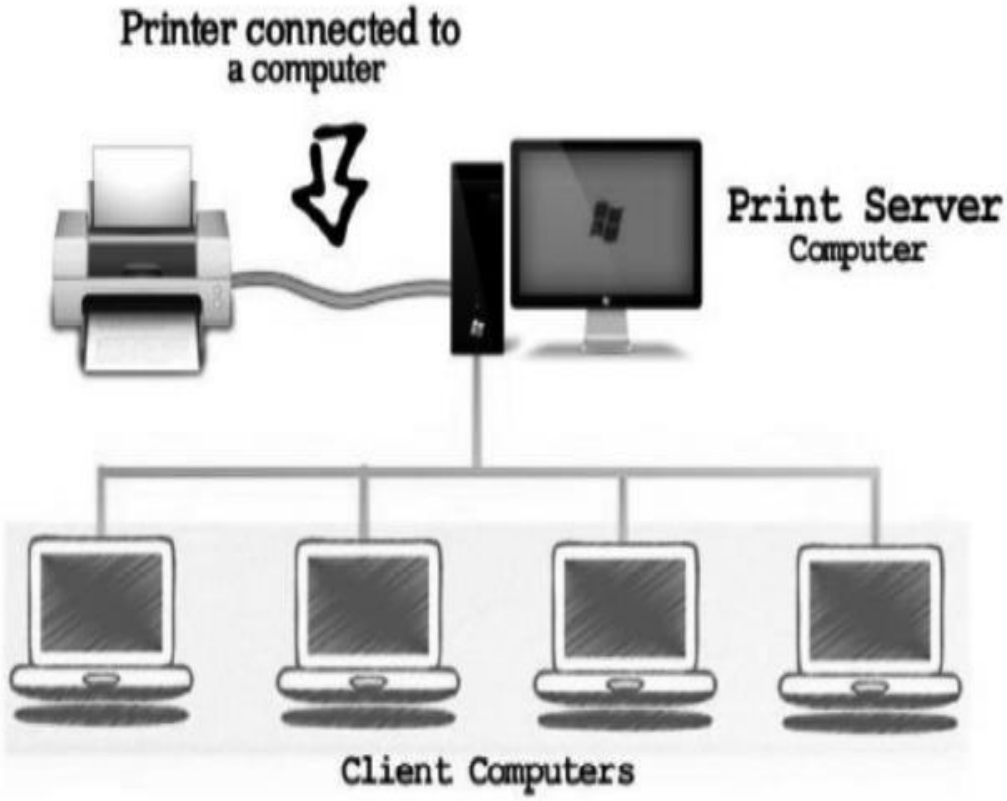
size = 5



Queue: Applications of Queue

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like .

This property of Queue makes it also useful When a resource is shared among multiple consumers.



Queue: Queue declaration.

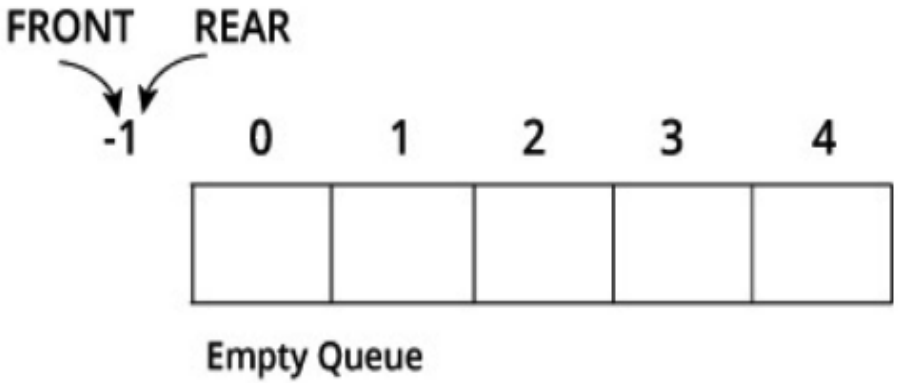
```
#define MAX_SIZE 5
#define DATA_TYPE int

typedef struct{
     DATA_TYPE elements[MAX_SIZE];
     int front;
     int rear;
}Queue;
```

- Queue functions prototypes

```
void Init(Queue *);
int IsFull(Queue *);
int IsEmpty(Queue *);
int Enqueue(DATA_TYPE, Queue *);
int Dequeue(DATA_TYPE *, Queue *);
int QueueSize(Queue *);
void ClearQueue(Queue *);
```

Queue: Queue Implementation.

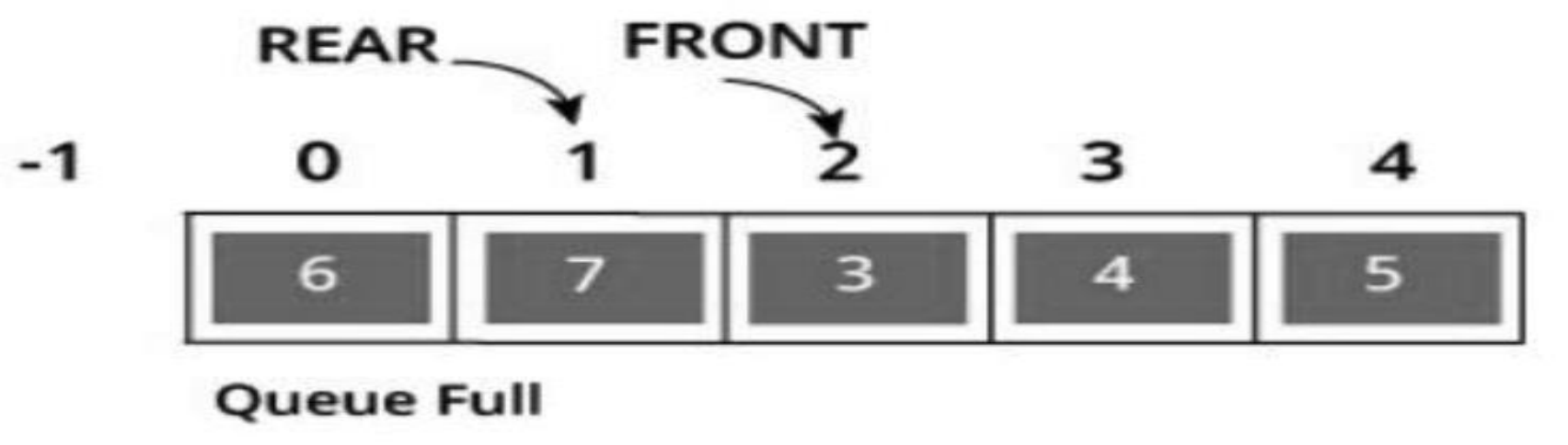
```
void Init(Queue *ptr_queue){

    ptr_queue->front = -1;
    ptr_queue->rear  = -1;
}
```




```
int IsEmpty(Queue *ptr_queue){

    if((ptr_queue->front == -1) && (ptr_queue->rear == -1)){
        return 1;
    } else {
        return 0;
    }
}
```

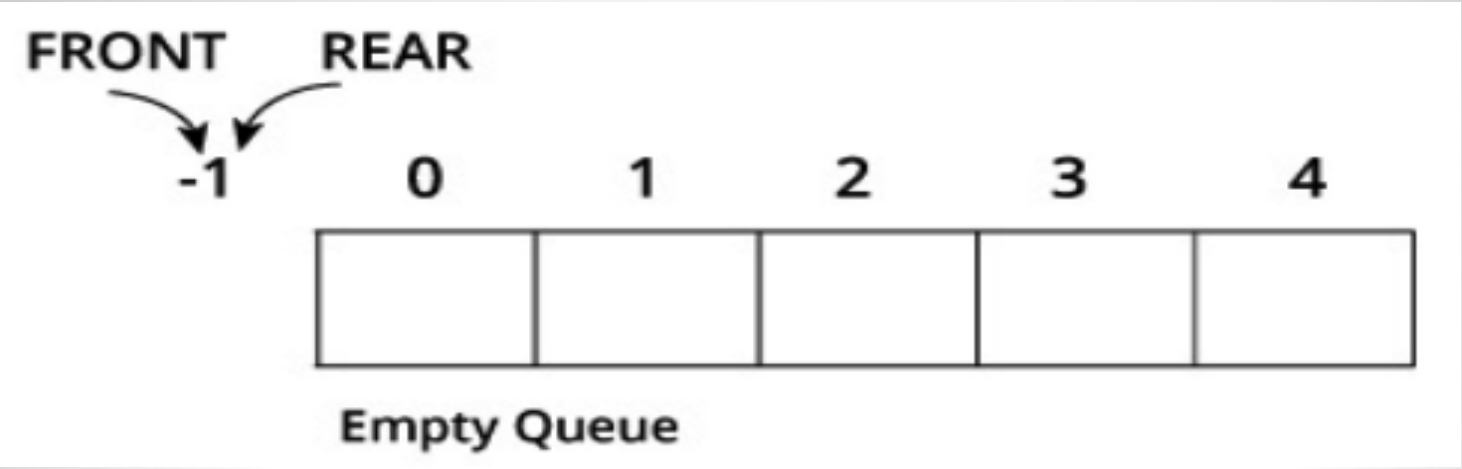
Queue: Queue Implementation.

```
int IsFull(Queue *ptr_queue){

    if((ptr_queue->rear + 1) % MAX_SIZE == ptr_queue->front){
        return 1;
    } else {
        return 0;
    }
}
```



Queue: Queue Implementation.

```
void ClearQueue(Queue *ptr_queue){

    ptr_queue->front = -1;
    ptr_queue->rear  = -1;
}
```

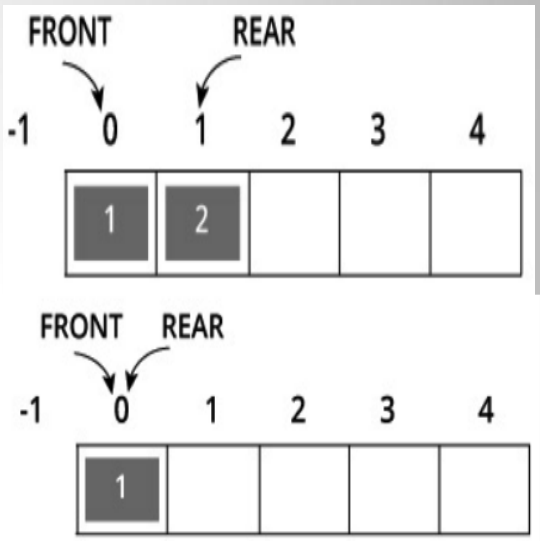


Queue: Queue Implementation.

```
int Enqueue(DATA_TYPE element, Queue *ptr_queue){

    // Is queue full?
    if((ptr_queue->rear + 1) % MAX_SIZE == ptr_queue->front){
        return 0;

    // Is queue empty?
    } else if ((ptr_queue->front == -1) && (ptr_queue->rear == -1)){
        ptr_queue->front = ptr_queue->rear = 0;

    } else {
        ptr_queue->rear = (ptr_queue->rear + 1) % MAX_SIZE;
    }

    ptr_queue->elements[ptr_queue->rear] = element;

    return 1;
}
```

Queue: Queue Implementation.

```
int Dequeue(DATA_TYPE *ptr_element, Queue *ptr_queue){

    // Is queue empty?
    if((ptr_queue->front == -1) && (ptr_queue->rear == -1)){

        return 0;
    }

    *ptr_element = ptr_queue->elements[ptr_queue->front];

    // if one element in queue
    if (ptr_queue->front == ptr_queue->rear){

        // reset queue
        ptr_queue->front = ptr_queue->rear = -1;

    } else {
        ptr_queue->front = (ptr_queue->front + 1) % MAX_SIZE;
    }
```



Queue: Queue Implementation.

```
int QueueSize(Queue *ptr_queue){

    if((ptr_queue->front == -1) && (ptr_queue->rear == -1)){// Is queue empty?

        return 0;

    } else if(ptr_queue->front < ptr_queue->rear){

        return (ptr_queue->rear - ptr_queue->front +1);

    } else if(ptr_queue->rear < ptr_queue->front){

        return ((MAX_SIZE - ptr_queue->front) + (ptr_queue->rear + 1));

    } else {

        return 1;
    }
```