ESPACE ENSEIGNEMENT CARDINAL MERCIER

Bases de la programmation

# Séance 7

# Structure de la mémoire et introduction aux pointeurs

*Sébastien Combéfis*

*mercredi 12 novembre 2014*

# Rappels du cours précédent

- Exercices pratique sur machine
- Prise en main de l'outil Code::Blocks
- Écriture, compilation et exécution d'un programme

# Variable en mémoire

- Une variable est caractérisée par :
  - Une adresse en mémoire
  - Un place occupée en mémoire (en octets)
  - Une valeur
  - Un nom symbolique se référant à l'adresse

```
int a = 17;
```

# Pointeur I

- Une pointeur est une variable qui stocke une adresse
- On déclare une variable de type pointeur avec $*$
- L'opérateur & permet d'obtenir l'adresse d'une variable

```
char c = 'a';
char *p = &c;
```

| | |
|---|---|
| a : 2000 | 'a' |
| p : 2004 | 2000 |

# Opérateur de déréférencement

- L'opérateur * permet de déréférencer un pointeur

  *Aller voir à l'adresse mémoire contenue dans le pointeur*

```
int a = 17;
int *p = &a;
int b = *p;
```

| | |
|---|---|
| a : 2000 | 17 |
| p : 2004 | 2000 |

# Pointeur II

- Un pointeur est également une variable

  *On peut donc stocker l'adresse d'un pointeur dans un pointeur*

```c
int a = 17;
int *p = &a;
int b = *p;

int **ap = &p;
```

| | |
|---|---|
| a : 2000 | 17 |
| p : 2004 | 2000 |
| ap : 2012 | 2004 |

# Le type pointeur

- Dans un type de variable, le $*$ désigne un pointeur
- La variable value est un pointeur vers un `int`

```
int *value;
```

- La variable value est un pointeur vers un pointeur de `int`

```
int **value;
```

# Le type pointeur

- Dans un type de variable, le $*$ désigne un pointeur
- La variable value est un pointeur vers un `int`

```
int *value;
```

- La variable value est un pointeur vers un pointeur de `int`

```
int **value;
```

# Le type pointeur

- Dans un type de variable, le $*$ désigne un pointeur
- La variable value est un pointeur vers un `int`

```
int *value;
```

- La variable value est un pointeur vers un pointeur de `int`

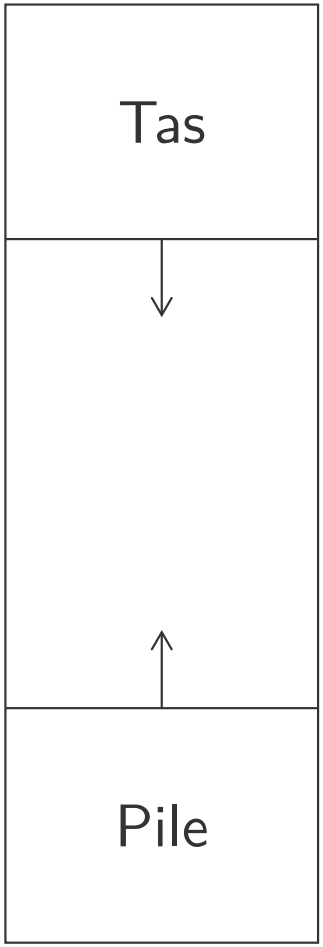
```
int **value;
```

# Mémoire dynamique I

- Il y a deux types de mémoire dans un programme
  - La pile contient les variables locales
  - Le tas contient les variables dynamiques
- Fonctions de gestion de la mémoire dans `stdlib.h`
  - On crée une nouvelle zone mémoire dynamique avec `malloc`
  - On libère la zone allouée avec `free`

```
int *p = malloc (sizeof (int));
free (p);
```

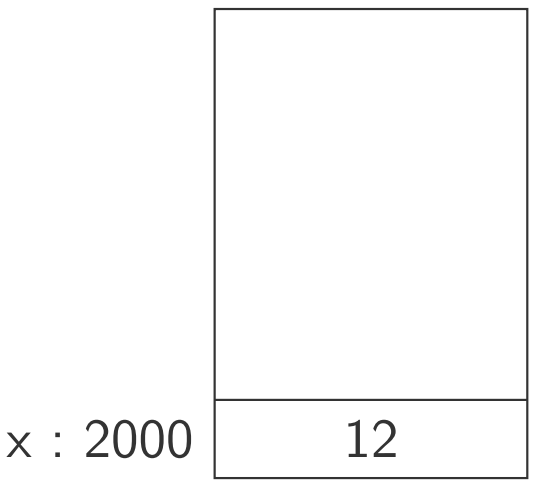
# Mémoire dynamique II




- La pile permet de gérer les appels de procédures/fonctions
- Le tas contient la mémoire dynamique

# Appel de procédure I

```
int main()
{
    int x = 12;
    printSum (x, -2);

    return 0;
}
```

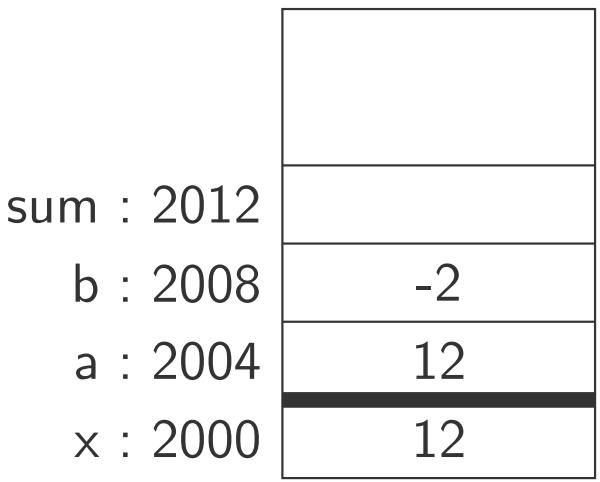


# Appel de procédure II

- Les paramètres de la procédure sont initialisés par l'appel

  printSum (x, −2);

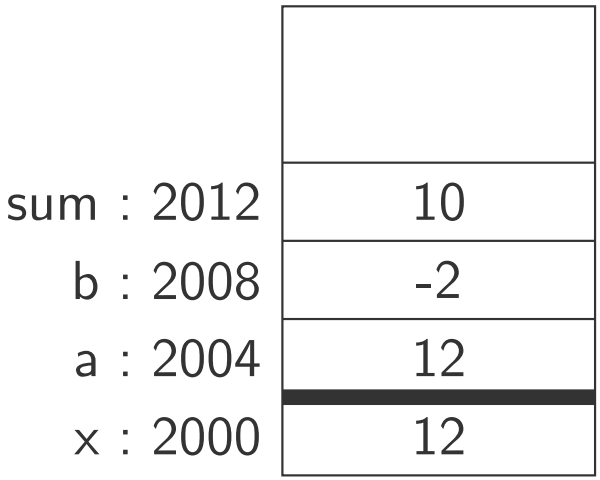
```
void printSum (int a, int b)
{
    int sum = a + b;
    printf ("%d + %d = %d\n", a, b, sum);
}
```



# Appel de procédure III

- Exécution de la première instruction

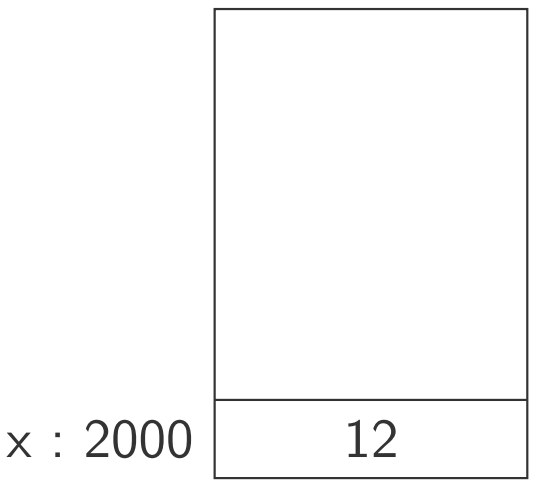
```
void printSum (int a, int b)
{
    int sum = a + b;
    printf ("%d + %d = %d\n", a, b, sum);
}
```



# Appel de procédure IV

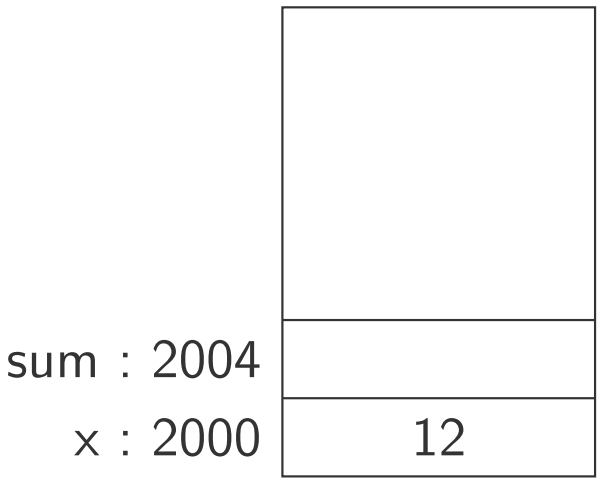
- Retour à la fonction main

```
int main()
{
    int x = 12;
    printSum (x, -2);

    return 0;
}
```



# Appel de fonction I

```
int main()
{
    int x = 12;
    int sum;
    sum = getSum (x, -2);

    return 0;
}
```



# Appel de fonction II

- Les paramètres de la fonction sont initialisés par l'appel

  getSum (x, −2);

```
int getSum (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

| | |
|---|---|
| sum : 2024 | |
| b : 2020 | -2 |
| a : 2016 | 12 |
| $ : 2008 | 2004 |
| sum : 2004 | |
| x : 2000 | 12 |

# Appel de fonction III

- Exécution de la première instruction

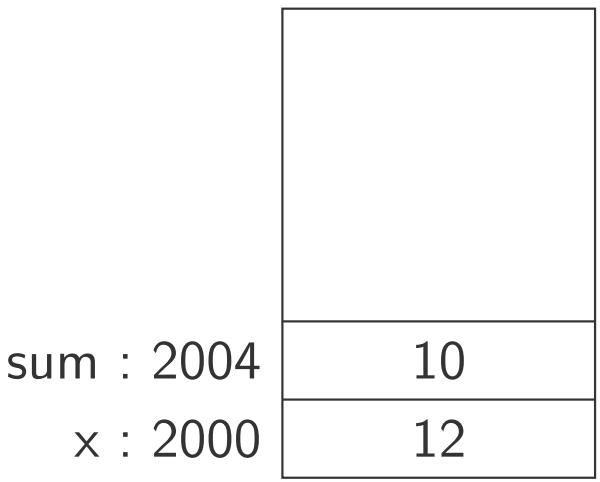
```
int getSum (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

| | |
|---|---|
| sum : 2020 | 10 |
| b : 2016 | -2 |
| a : 2012 | 12 |
| $ : 2008 | 2004 |
| sum : 2004 | |
| x : 2000 | 12 |

# Appel de fonction IV

- Retour à la fonction main

```
int main()
{
    int x = 12;
    int sum;
    sum = getSum (x, -2);

    return 0;
}
```



# Échanger les valeurs de deux variables I

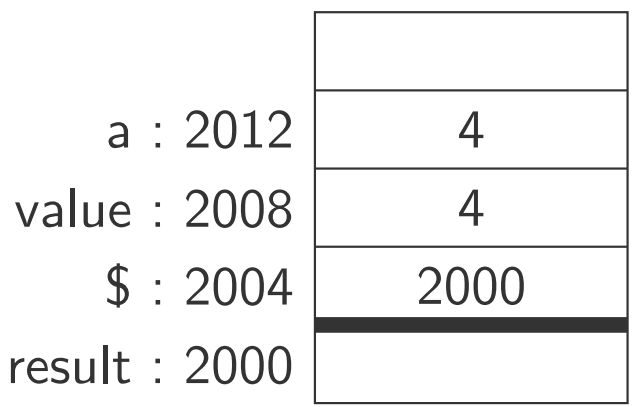
```c
void swap (int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void main()
{
    int x = 2;
    int y = 15;
    swap (x, y);
    printf ("x = %d et y = %d\n", x, y);

    return 0;
}
```

# Échanger les valeurs de deux variables II

```c
void swap (int *pa, int *pb)
{
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

void main()
{
    int x = 2;
    int y = 15;
    swap (&x, &y);
    printf ("x = %d et y = %d\n", x, y);

    return 0;
}
```

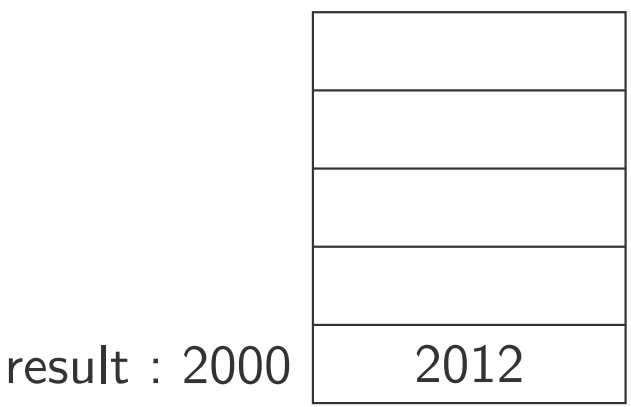
# Renvoyer un pointeur I

```c
int* newInt (int value)
{
    int a = value;
    return &a;
}

int main()
{
    int *result = newInt (4);
    printf ("%d\n", *result);
    return 0;
}
```



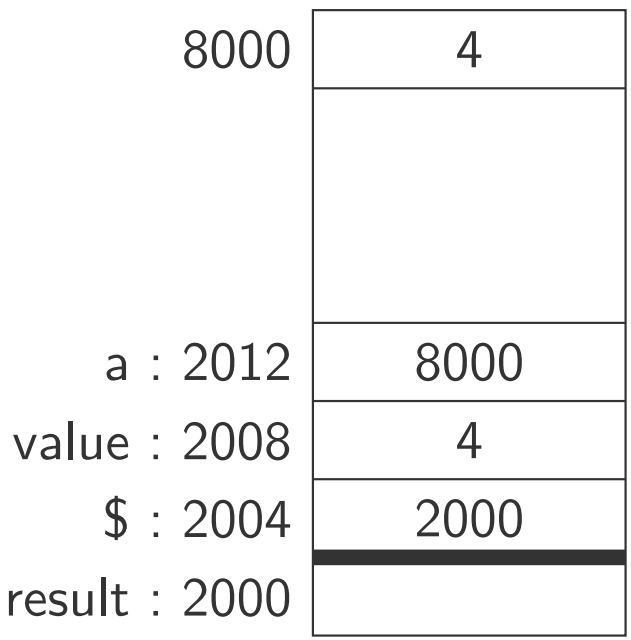
# Renvoyer un pointeur II

```c
int* newInt (int value)
{
    int a = value;
    return &a;
}

int main()
{
    int *result = newInt (4);
    printf ("%d\n", *result);
    return 0;
}
```



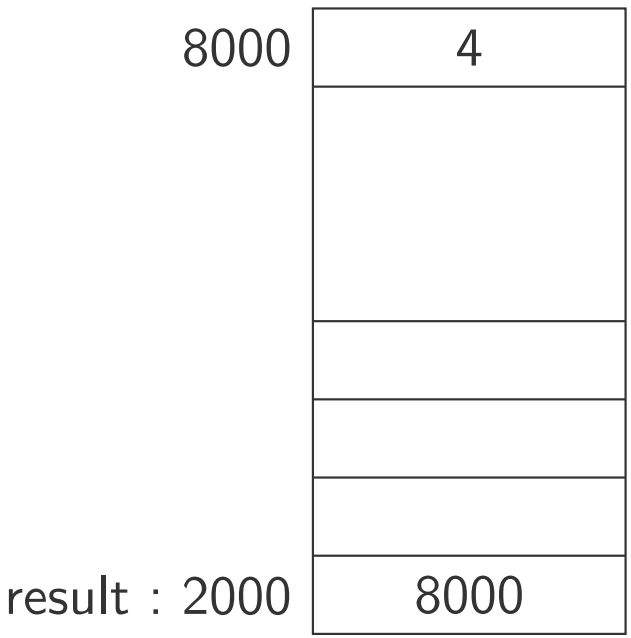
# Renvoyer un pointeur III

```
int* newInt (int value)
{
    int *a = malloc (sizeof (int));
    *a = value;
    return a;
}
```



# Renvoyer un pointeur IV

```c
int main()
{
    int *result = newInt (4);
    printf ("%d\n", *result);

    return 0;
}
```



# Exercice

| | |
|---|---|
| 12000 | 42 |
| | |
| 9016 | 18 |
| 9012 | -4 |
| 9008 | 2 |
| 9004 | 8 |
| 9000 | -5 |
| | |
| 8004 | 9000 |
| 8000 | 12000 |
| | |
| tab : 2000 | 8000 |

Sachant qu'on a `int **tab`

1. `tab`
2. `*tab`
3. `**tab`
4. `&tab`