Langage C avancé
Les Pointeurs

Samuel KOKH

samuel.kokh@cea.fr

MACS 1 – Institut Galilée

Variables et mémoire

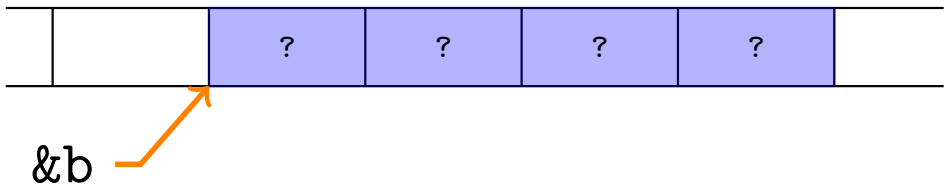
Que fait la machine ?

```
void fct(void){
   int b ;
   ...

   b = 3 ;
   ...
}
```

Variables et mémoire

```
int b ;
```

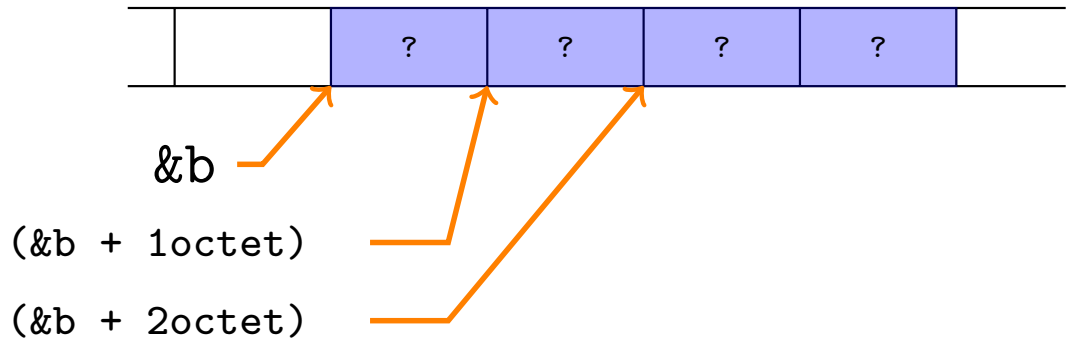
- la machine réserve une zone de mémoire de longueur `sizeof(int)` (4 octets).
- Cette zone mémoire démarre à l'octet «numéroté» `&b`, ou octet d'adresse `&b`.



Variables et mémoire

```
int b ;
```

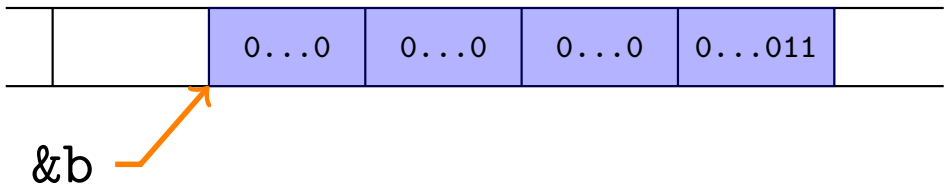
- la machine réserve une zone de mémoire de longueur `sizeof(int)` (4 octets).
- Cette zone mémoire démarre à l'octet «numéroté» `&b`, ou octet d'adresse `&b`.



Variables et mémoire

```
b = 3 ;
```

La machine écrit la séquence binaire qui correspond à 3 sur une longueur de `sizeof(int)` en commençant à écrire à l'adresse `&b`.



Variables et mémoire

```
void fct(void){
   int b ;
   ...
   b = 3 ;
   ...
}
```

Gestion de la durée de vie des variables

Jusqu'ici . . .

- totalement gérée par le programme. La vie d'une variable s'arrête hors de son bloc de code
- c'est automatique (cf. utilisation du mot-clef `auto`).

Pas de contrôle sur la pérénnité des variables et de leur valeurs...

Persistance de la mémoire

Peut-on écrire une fonction qui permet d'échanger les valeur de deux variables ?

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack : appels des fonctions

Stack associé au programme

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de printf()
point de retour
espace pour vars interne
...
espace pour le paramètre
char[]=“hello”

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de Swap()
point de retour
espace pour vars interne
int tmp = 0;
espace pour les paramètres
int a = 3;
int b = 4;

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de Swap()
point de retour
espace pour vars interne
int tmp = 3;
espace pour les paramètres
int a = 3;
int b = 4;

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de Swap()
point de retour
espace pour vars interne
int tmp = 3;
espace pour les paramètres
int a = 4;
int b = 4;

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de Swap()
point de retour
espace pour vars interne
int tmp = 3;
espace pour les paramètres
int a = 4;
int b = 3;

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de printf()
point de retour
espace pour vars interne
...
espace pour les paramètre
char[]=“bye”

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Frame de main()
espace pour vars interne :
n = 3;
p = 4;

Stack : appels des fonctions

```c
#include <stdio.h>
void Swap(int a, int b){
    int tmp = 0;
    tmp = a;
    a = b;
    b = tmp;
}


int main(void){
    int n = 3;
    int p = 4;
    printf("hello\n");
    Swap(n,p);
    printf("bye\n");
    return 0;
}
```

Stack associé au programme

Exécution terminée

Que faire ?

Notion de pointeur en C

La notion de pointeur permet de

- manipuler les adresses des objets en mémoire
- manipuler les objets grâce à leur adresse

On verra par la suite que cela aussi permet de gérer la mémoire de manière dynamique...

Deux opérateurs importants

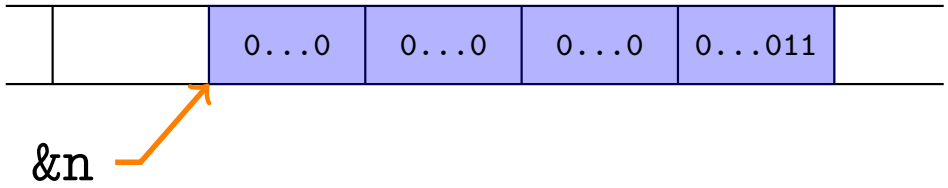
opérateur &

Permet d'obtenir l'adresse d'un objet

opérateur *

Permet d'accéder à un objet (ou à une zone mémoire) connaissant son adresse.

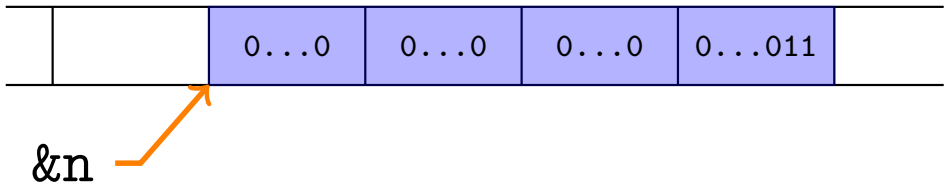
Opérateur &




```
int n=3;
printf("adresse de n = %p \n",&n);
```

- `&n` est une valeur entière. Elle indique la position du 1er octet qui sert à coder la valeur de `n` en mémoire
- la machine sait que ce nombre désigne une zone de stockage de taille `sizeof(int)`

Opérateur *




```
int n=3;
*(&n) = 4;
printf("valeur de n = %d \n",n);
```

Si `A` désigne l'adresse d'une zone mémoire (de taille L)

- `*A` permet d'accéder **en lecture** à la zone mémoire démarre à l'adresse A
- `*A` permet d'accéder **en écriture** à la zone mémoire démarre à l'adresse A

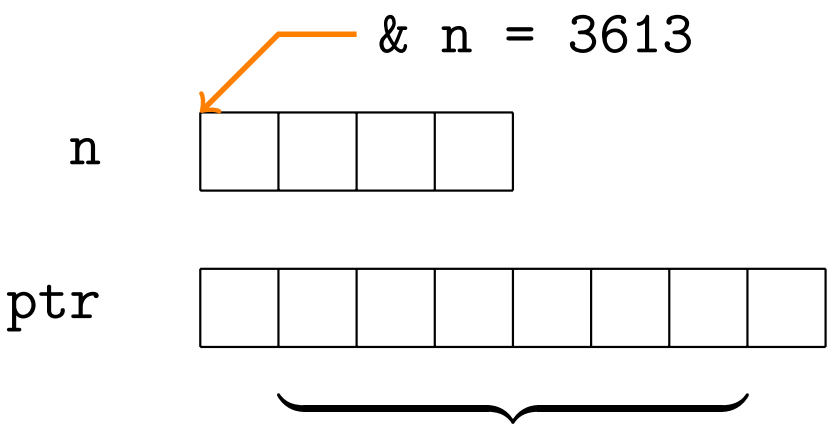
La notion de pointeur

Un pointeur est une variable entière qui porte deux informations

- l'adresse de la zone mémoire (valeur entière) de stockage d'une variable d'un type donné
- la taille du type de l'objet stocké

Un pointeur est un objet «typé» (pointeur sur...).

```
int n=3;
int* ptr; // ptr est un pointeur sur int

ptr = &n; // la valeur de ptr est l'adresse de n
```



séquence de bits qui codent 3613

```
int n=3;
int* ptr; // ptr est un pointeur sur int

ptr = &n; // la valeur de ptr est l'adresse de n
```

De manière générale :

```
1 T var;  // var de type T
2 T* ptr; // pointeur sur T
3 ptr = &var;
4 *ptr = <valeur> ; // si cette operateur a un sens pour l
```

exemple

```
1 float f = 3.2;     // var de type float
2 float* pF;         // pointeur sur float
3 pF = &f;
4 *pF = 5.1 ;
```

Adresse NULL

Adresse interdite

Il existe une adresse pour laquelle le C garantit que l'on ne pourra jamais y faire démarrer le stockage d'une valeur. C'est l'adresse `NULL`.

`NULL` est une valeur (souvent 0), définie par la librairie standard, dans le fichier `stddef.h` grâce à une intruction préprocesseur (macro).

ATTENTION

`NULL` (adresse interdite) $\neq$ `NUL` (caractère de terminaison, '\0')

Pourquoi utiliser NULL ?

exemple

```
float* pF = NULL;
```

Tant que pF n'est pas reçu de valeur valide, toute tentative d'affectation à *pF va causer un plantage.

MIEUX VAUT PLANTER LE PROGRAMME QUE LAISSER LE PROGRAMME TOURNER EN UTILISANTS DES ADRESSES ERONNEES !

Solution pour la fonction Swap

La fonction qui permet de réaliser l'échange des valeurs de deux variables doit prendre comme argument des pointeurs sur les variables en question.

```
1 void Swap(int* a, int* b){
2    int tmp = 0;
3    tmp = *a;
4    *a = *b;
5    *b = tmp;
6 }
```

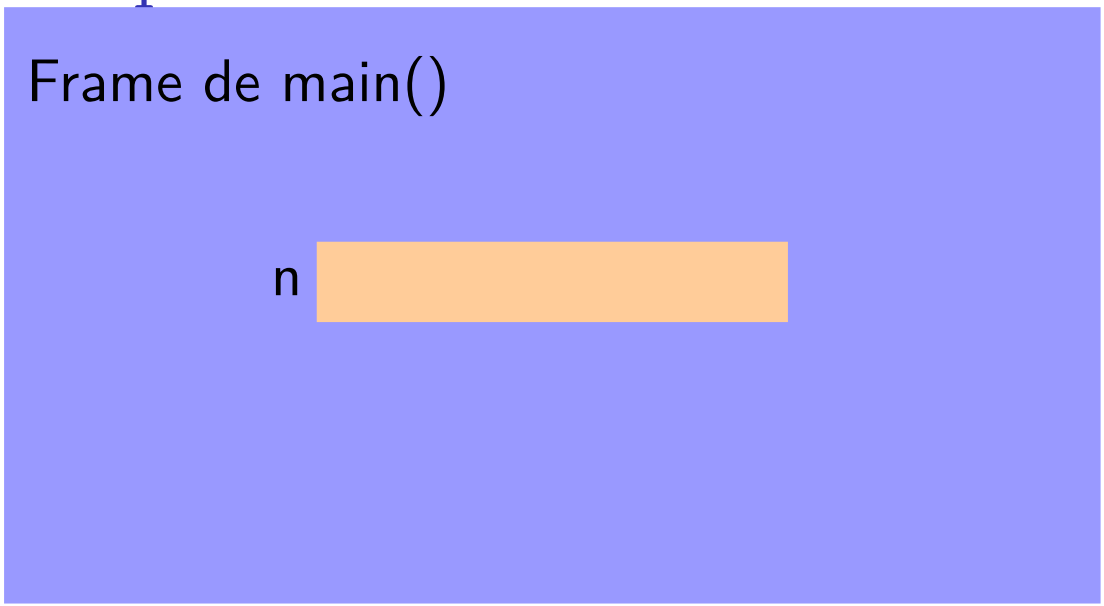
Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```

Frame de main()

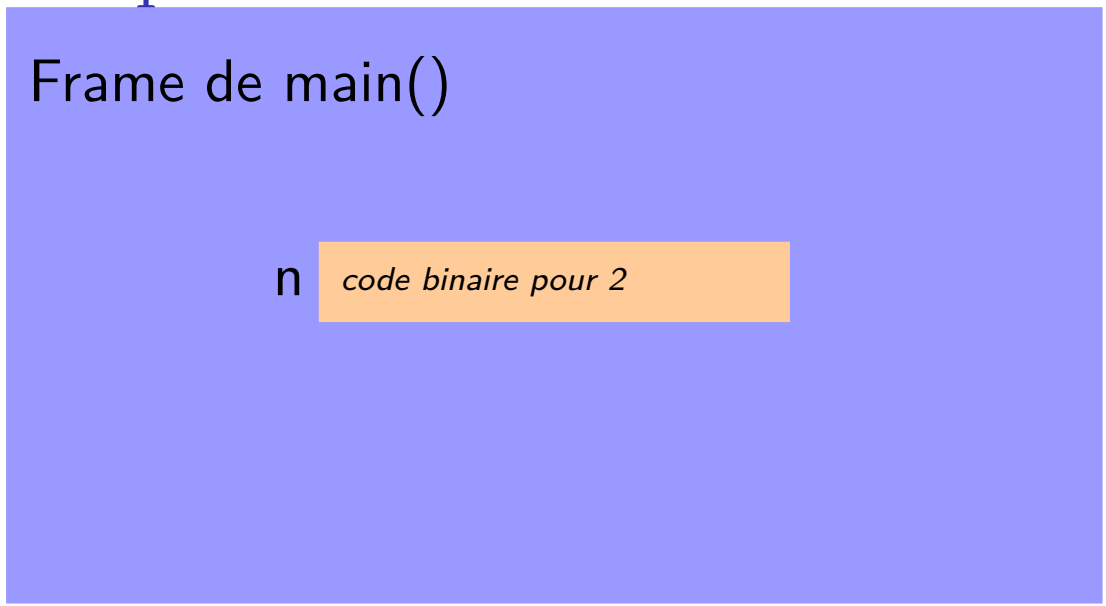
Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



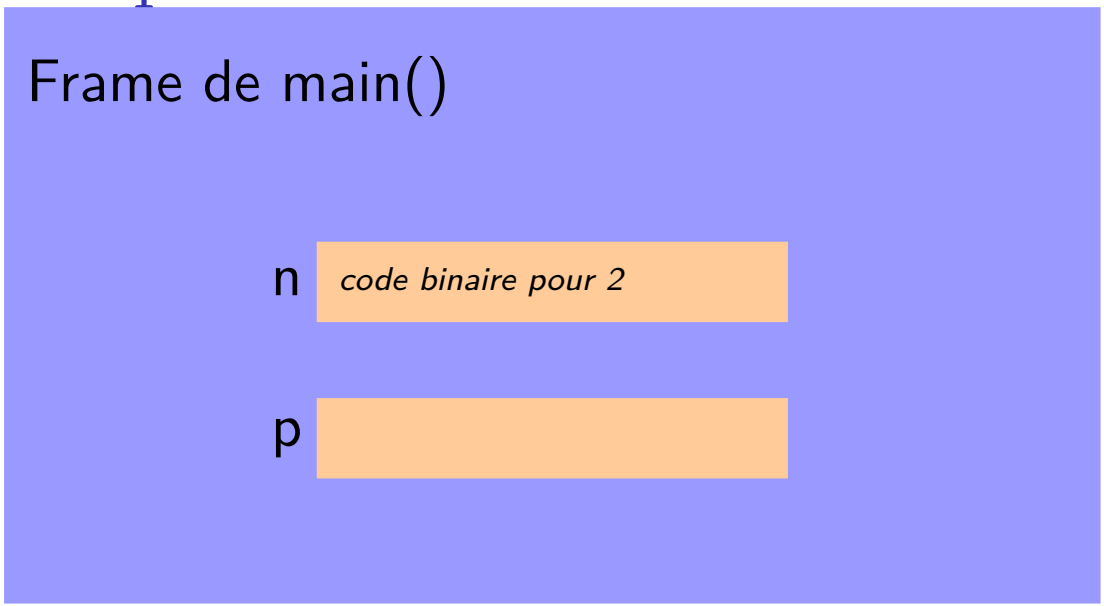
Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



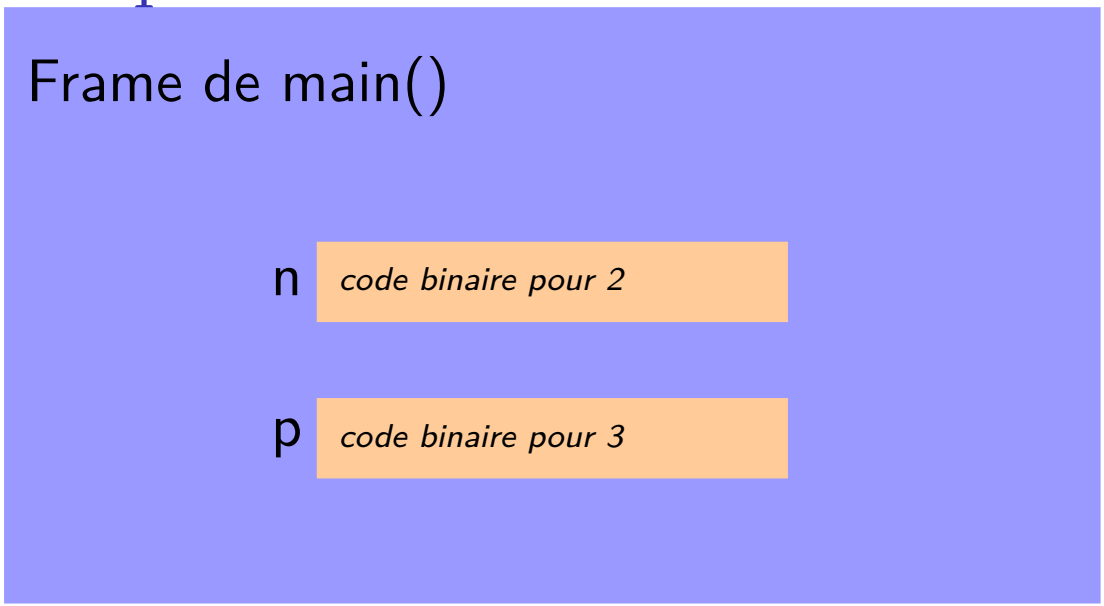
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



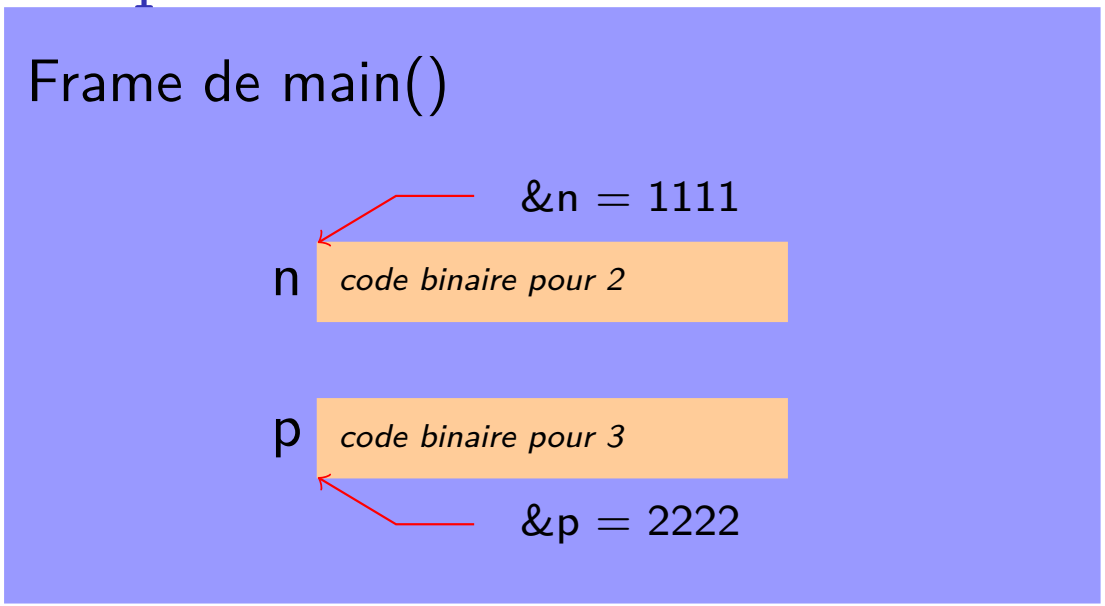
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



Solution pour la fonction Swap

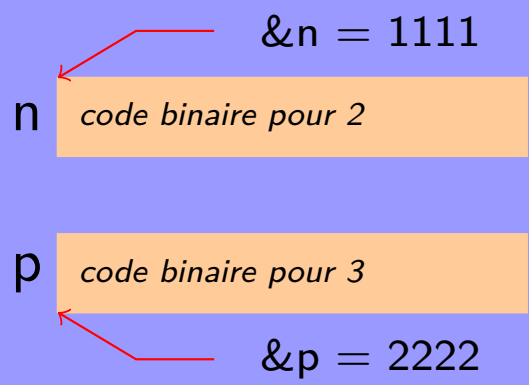
```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```

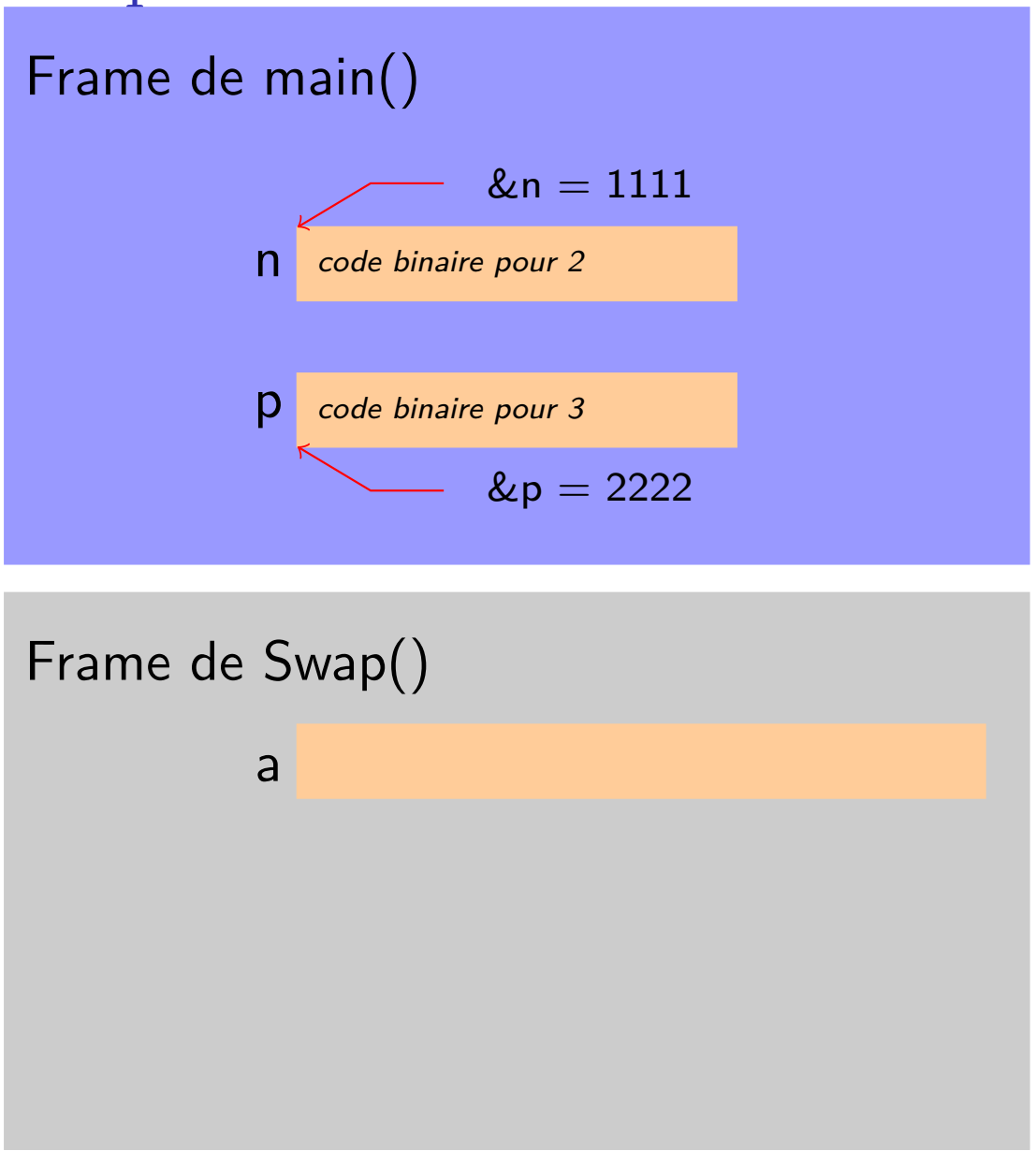
Frame de main()



Frame de Swap()

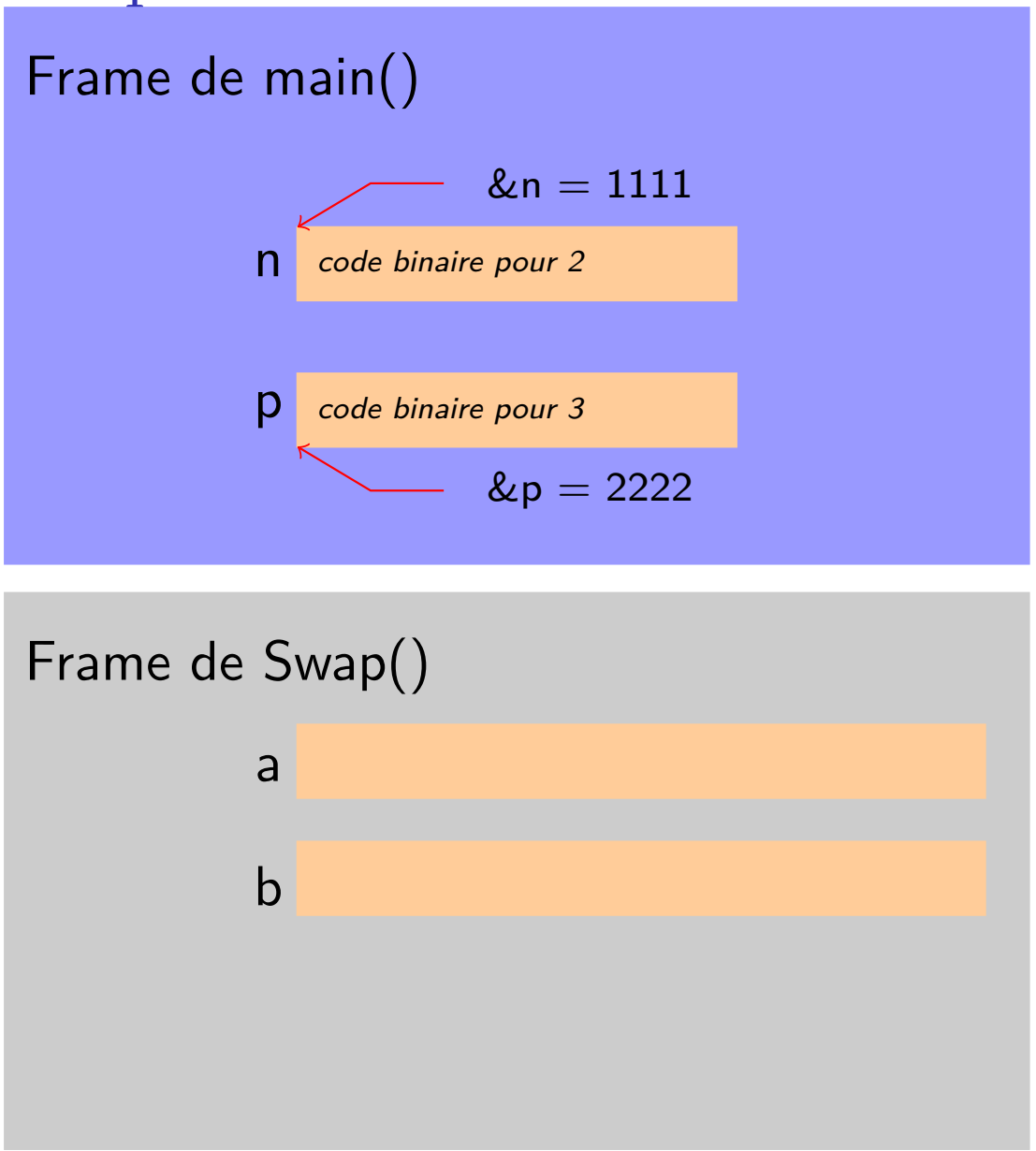
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



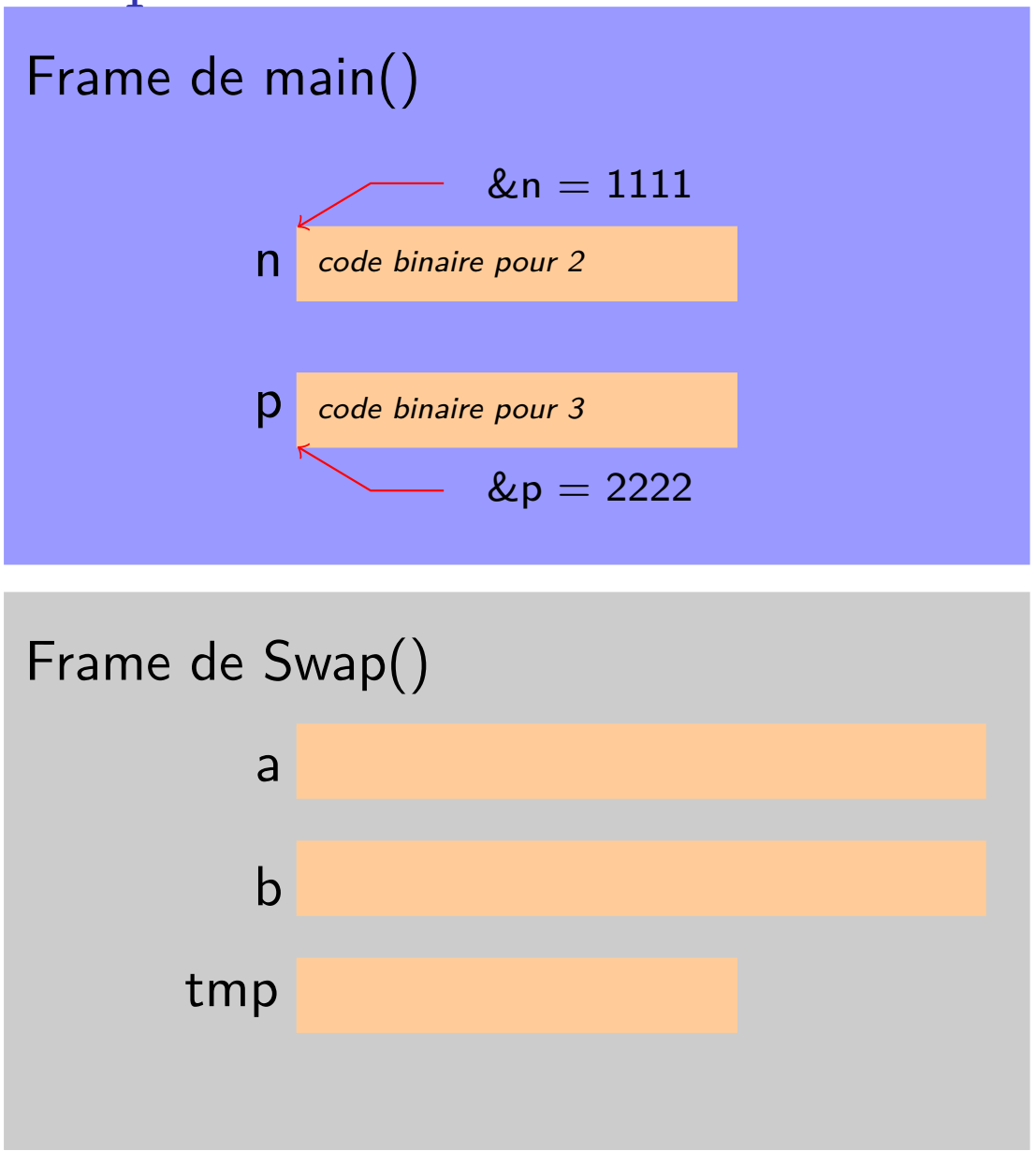
Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



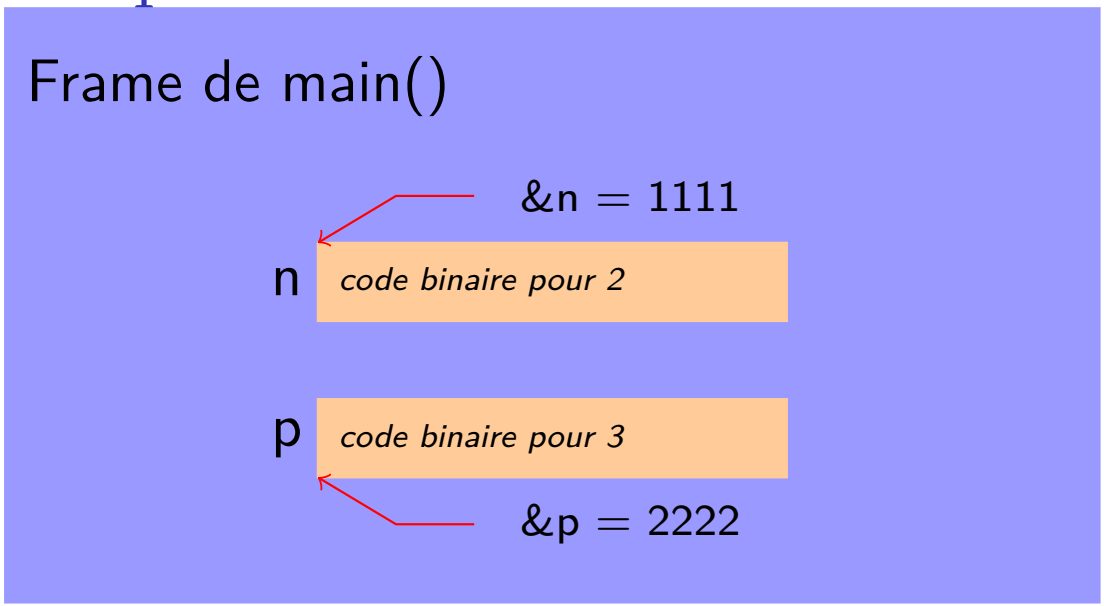
Solution pour la fonction Swap

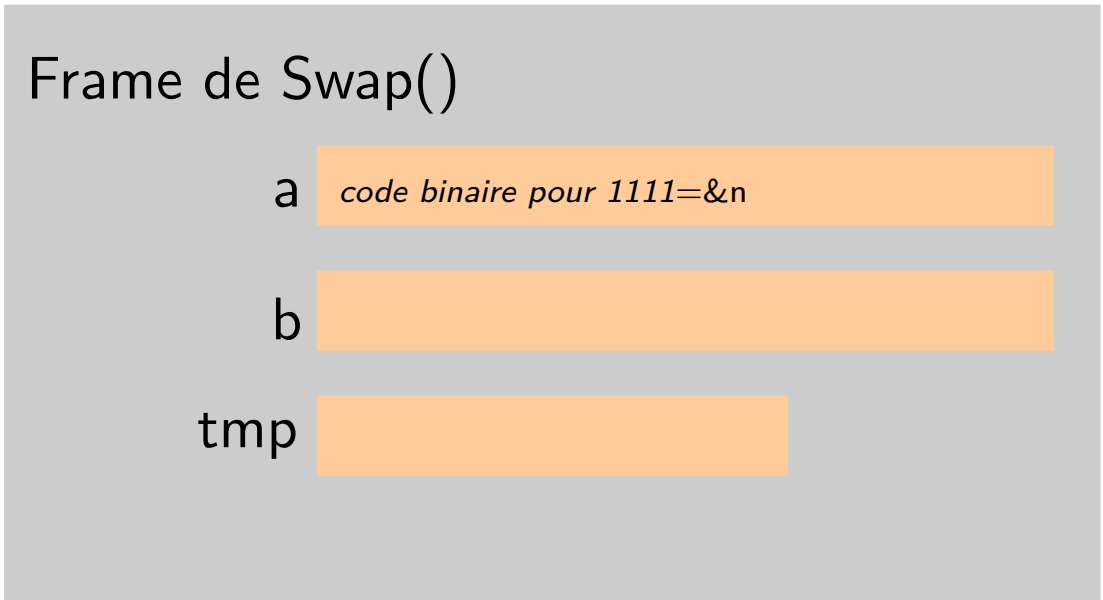
```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



Solution pour la fonction Swap

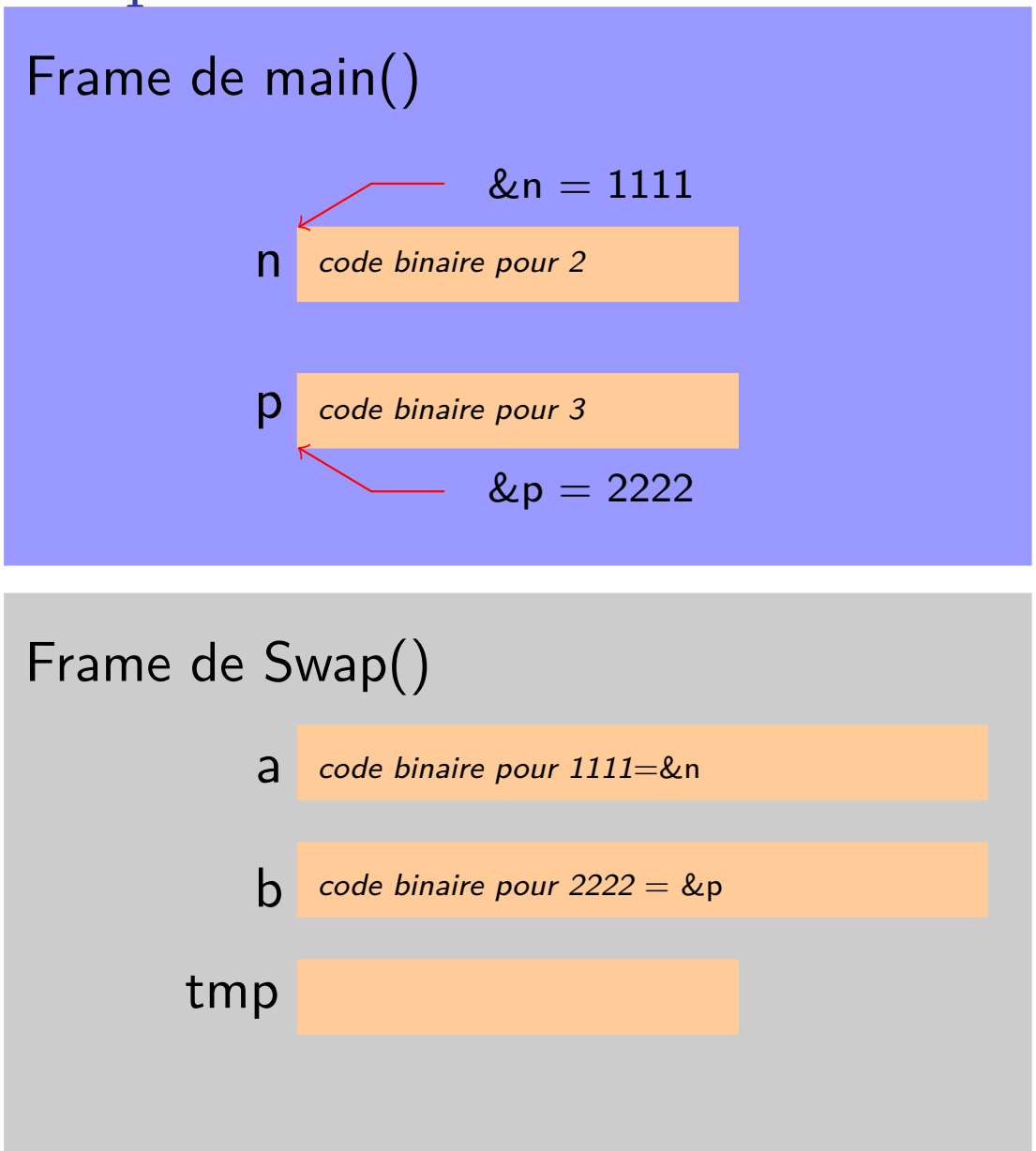
```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```





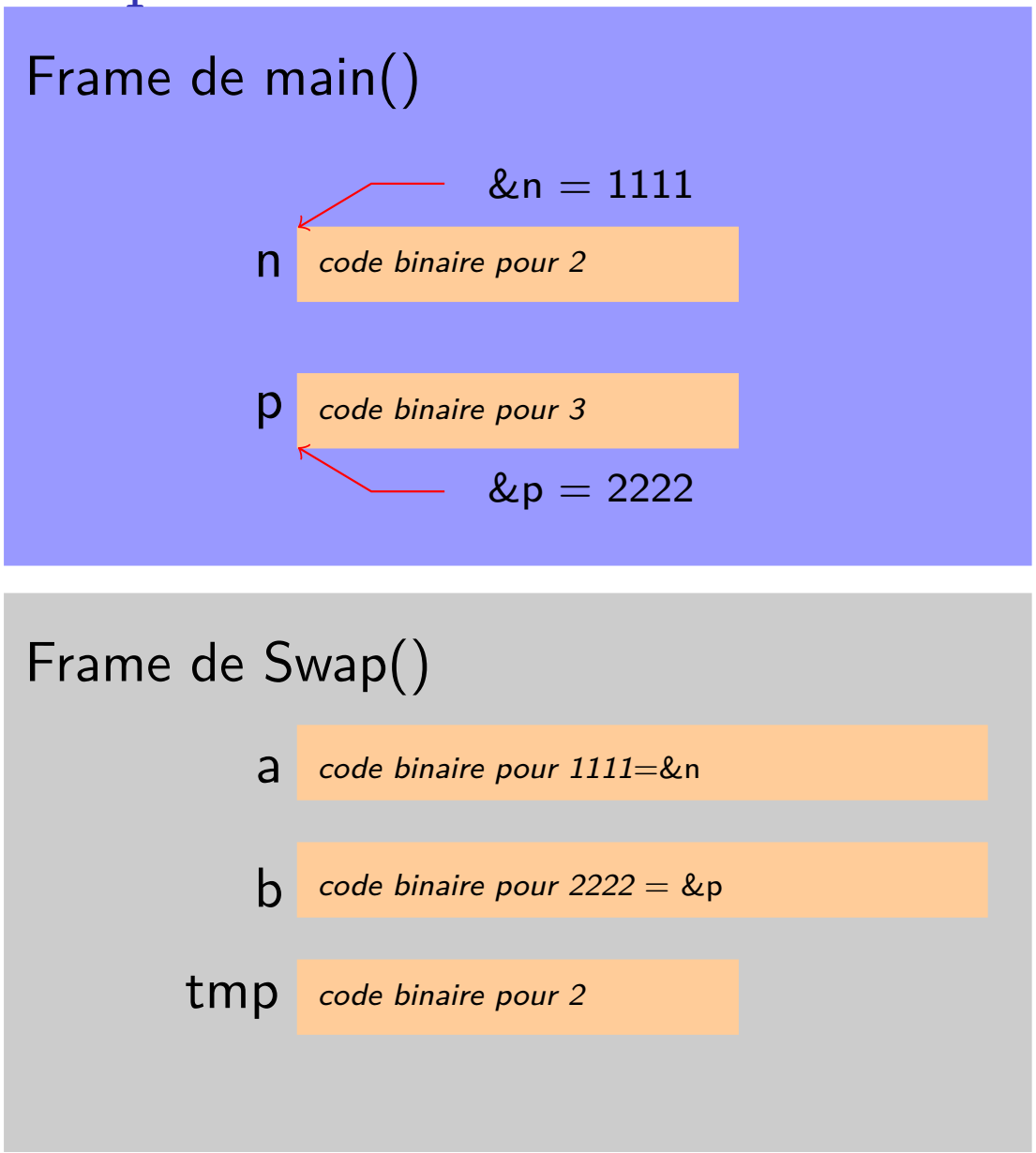
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



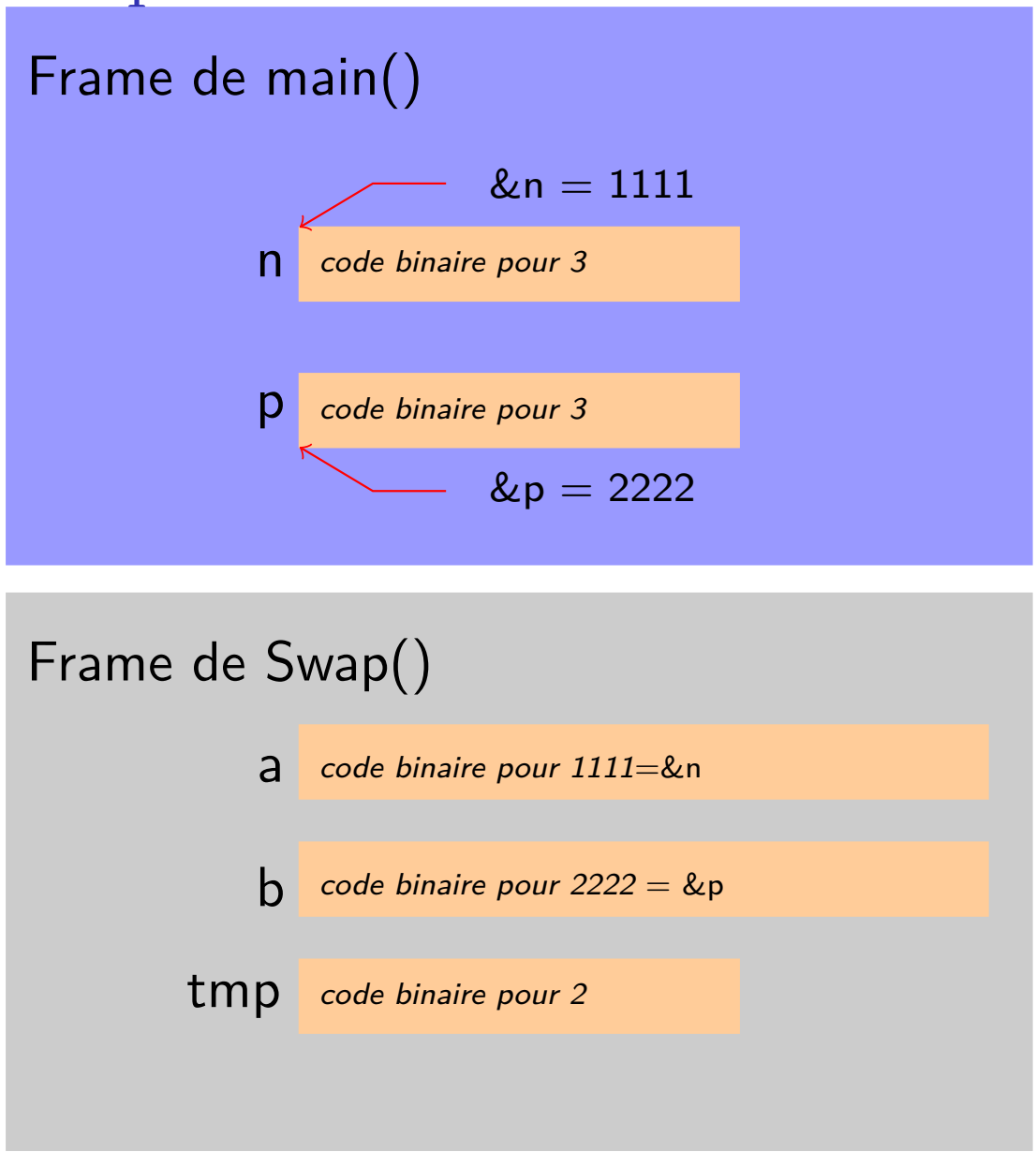
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



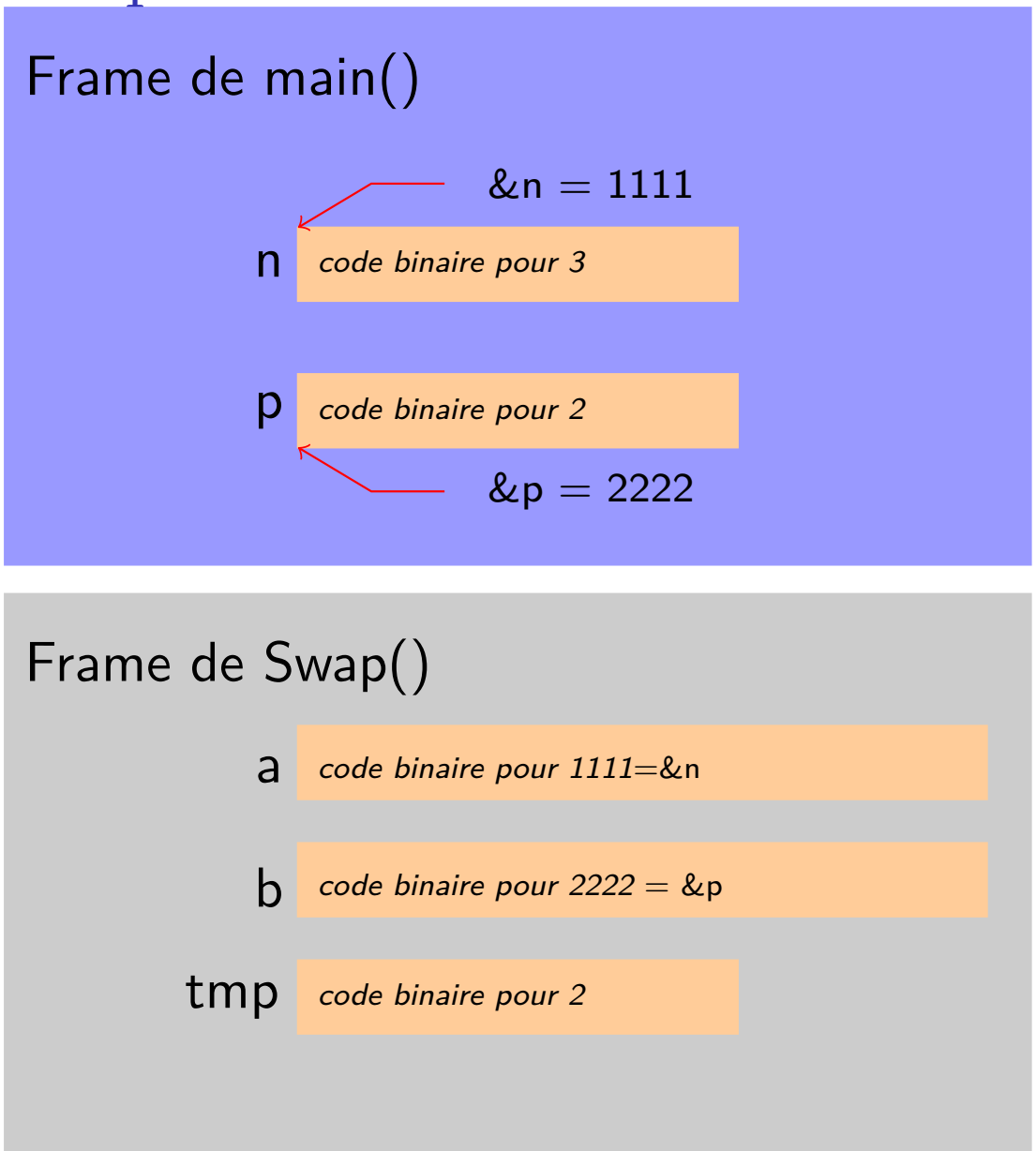
Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



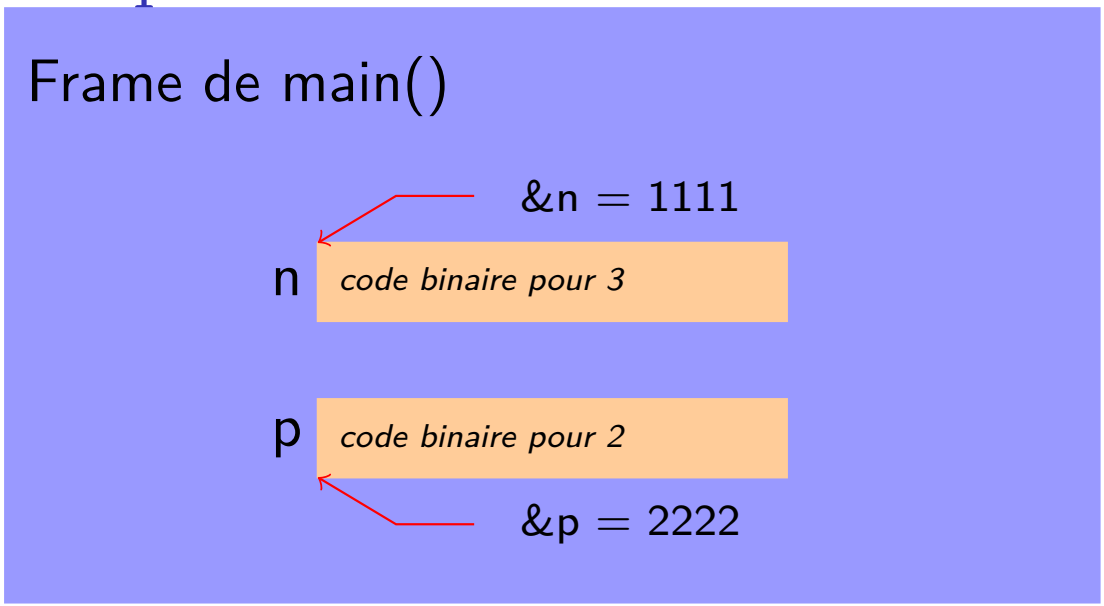
Solution pour la fonction Swap

```
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



Solution pour la fonction Swap

```c
void Swap(int* a, int* b){
  int tmp = 0;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



int main(void){
  int n = 2;
  int p = 3;

  Swap(&n, &p)
  return 1;
}
```



Arithmetique de pointeurs

```
1 int* ip;      // ip de type (int*)
2 short* sp;    // sp de type (short*)
3 char* chp;    // chp de type (char*)
```

Différence supplémentaire entre les types de pointeurs

Ces types permettent tous de stocker des adresses, mais :

- l'opérateur $*$ tient compte de la taille des objets stockés
- l'opération d'addition tient compte de la taille des objets stockés

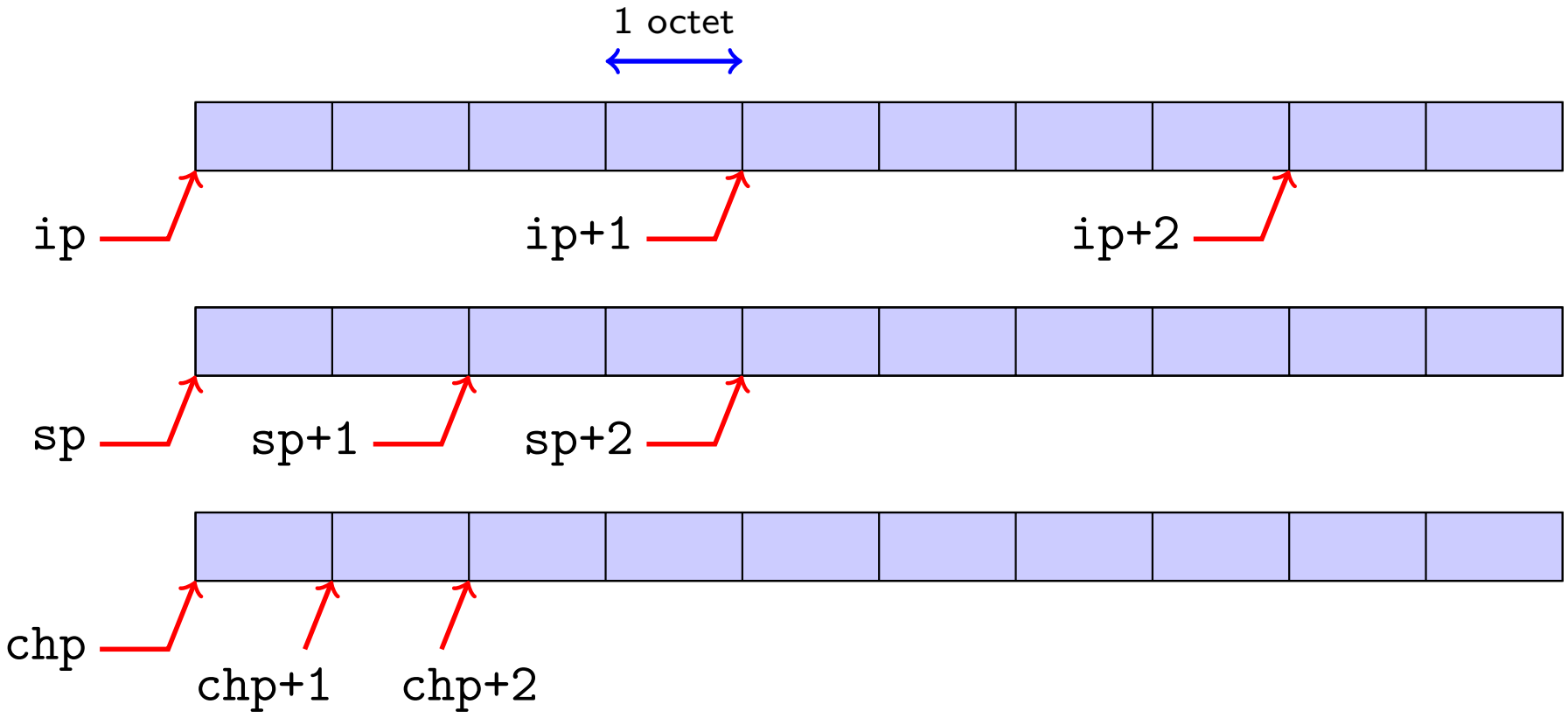
Arithmetique de pointeurs

```
1 int* ip;      // ip de type (int*)
2 short* sp;    // sp de type (short*)
3 char* chp;    // chp de type (char*)
```

- `ip+1` pointe vers l'adresse mémoire décalée de la taille d'un `int` (4 octets) à partir de `ip`
- `sp+1` pointe vers l'adresse mémoire décalée de la taille d'un `short` (2 octets) à partir de `sp`
- `chp+1` pointe vers l'adresse mémoire décalée de la taille d'un `ch` (1 octet) à partir de `chp`

Arithmetique de pointeurs

```
1 int* ip;      // ip de type (int*)
2 short* sp;    // sp de type (short*)
3 char* chp;    // chp de type (char*)
```



Les pointeurs : à propos de la syntaxe du C et ses ambiguïtés

Déclaration d'un pointeur

Les trois lignes de code ci-dessous sont également juste en terme de syntaxe et réalise la même opération en C.

```
1.  int  *ptr;
2.  int*  ptr;
3.  int  *  ptr;
```

Les pointeurs : à propos de la syntaxe du C et ses ambiguïtés

Déclaration d'un pointeur

Les trois lignes de code ci-dessous sont également juste en terme de syntaxe et réalise la même opération en C.

1. `int *ptr;` $\longrightarrow$ l'objet `*ptr` est de type `int`
2. `int* ptr;` $\longrightarrow$ l'objet `ptr` est de type `(int*)`
3. `int * ptr;` $\longrightarrow$ notation « neutre »

Pointeurs et déclarations multiples

Les trois lignes suivantes sont correctes

1. `int *ptr, n;`
2. `int* ptr, n;`
3. `int * ptr, n;`

elles sont équivalentes à

```
int *ptr;
int n;
```

La syntaxe la plus consistante semble être (dans ce cas) la numéro 1.

Les trois initialisations suivantes sont syntaxiquement exactes.

1.

```
int  n;
int  *ptr=&n;
```

2.

```
int  n;
int*  ptr=&n;
```

3.

```
int  n;
int  *  ptr=&n;
```

Elles sont toutes trois équivalentes à

```
int  n;
int  *  ptr;
ptr = &n;
```

L'objet `&n` est de type `(int*)` et donc la syntaxe la plus consistante semble (dans ce cas) être la numéro 2.

De même...

Les trois lignes suivantes sont correctes

```
1.  int  *ptr = NULL;
2.  int*  ptr = NULL;
3.  int  *  ptr = NULL;
```

elles sont équivalentes à

```
int  *ptr;
ptr = NULL;
```

L'objet `NULL` est de type `(void*)` et donc la syntaxe la plus consistante semble être (dans ce cas) la numéro 2.

Type Pointeur Générique : `void*`

Deux choses essentielles derrière la notion de pointeur sur variable

```
T *ptr;
```

`(T*)` {

- adresse mémoire : contenue dans `ptr`
- la taille de l'objet qui démarre à l'adresse indiquée : `sizeof(T)`

Type Pointeur Générique : `void*`

Deux choses essentielles derrière la notion de pointeur sur variable

```
int *ptr;
```

`(int*)` {

- adresse mémoire : contenue dans `ptr`
- la taille de l'objet qui démarre à l'adresse indiquée : `sizeof(int)`

Type Pointeur Générique : `void*`

Deux choses essentielles derrière la notion de pointeur sur variable

```
int *ptr;
```

`(int*)` {

- adresse mémoire : contenue dans `ptr`
- la taille de l'objet qui démarre à l'adresse indiquée : `sizeof(int)`

Type Pointeur Générique (`void*`)

Il existe un type particulier de pointeur qui ne tient compte que de l'adressage.

C'est le type pointeur générique (`void*`).

Type (void*) : pas d'info. sur la taille des objets vers lesquels les adresses pointent.

Dans ce cas, comment se comportent

- l'opération d'addition de variable (void*) (arithmétique de pointeur) ?
- l'opération de déréférencement * ?

Arithmétique de pointeur pour (void*)

```
int n=3;
int* p=&n;
void* ptr = &n;

p = p + 1 ;      // decale l'adresse stockee de   sizeof(int) octets
p = p + 5 ;      // decale l'adresse stockee de 5*sizeof(int) octets
ptr = ptr + 1 ; // decale l'adresse stockee de              1  octet
ptr = ptr + 5 ; // decale l'adresse stockee de              5  octets
```

Opération * pour un pointeur (void*)

Interdit par le langage C !

Un pointeur générique seul

- désigne le début d'une zone mémoire
- ne permet pas d'accéder à cette zone : ni en écriture, ni en lecture via l'opérateur * ! On ne peut pas déréférencer un pointeur générique.

```
double d=2.1;
double e = 0.1;
void* ptr = &d;   // ptr pointe vers d

*ptr = 3.14;                              // interdit!
e = *ptr                                  // interdit!
printf("resultat = %g\n", *ptr);          // interdit!
```

Comment accéder en lecture/écriture à une adresse pointée par un pointeur générique ?

On passe par un pointeur typé ! C'est un typage forcé.

```
void *ptr ;
*ptr = 3 ; //interdit !
```

```
void *ptr ;
int n = 3 ;
*ptr = &n ;
*ptr = 3 ; //interdit !
```

```
void *ptr ;
int n = 3 ;
ptr = &n ;
*( (int*) ptr ) = 3; //ok !
```

```
void fct (void *ptr) {
    int *p;
    p = ptr;
    *p = 3;   //ok !
    ...
}
```

Intérêt des pointeurs génériques ?

Programmation générique en langage : on peut programmer des routines génériques en s'affranchissant des types.

La fonction

```
void *memcpy(void *dest, const void *src, size_t n)
```

recopie à l'adresse `dest` une zone mémoire de `n` octets commençant à l'adresse `src`.

Version générique de la fonction d'échange des valeurs de deux variables

?

Version générique de la fonction d'échange des valeurs de deux variables

```
void SwapVar (void* a, void* b, size_t size){
    char* buf = malloc(size);
    memcpy(buffer, a, size); // copie   a => buf
    memcpy(a, b, size);      // copie   b => a
    memcpy(b, buf, size);    // copie buf => b
    free(buf);
}
```

```
int a = 3;
int b = 2;
Swap(&a, &b, sizeof(int));
```