Langage C avancé : Séance 5

Arguments fournis à l'exécution

Fonctions récursives

Pointeurs de fonction

Librairies statiques et dynamiques

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Arguments fournis à l’exécution

Arguments fournis à l'exécution

Il est possible dans un code C, de donner la possibilité à l'utilisateur du binaire de fournir des valeurs à l'exécution.

Pour cela, la fonction principale (main) peut s'écrire de 2 façons différentes:

```
int main()
{
  …
}
```

```
int main(int argc, char **argv)
{
  …
}
```

Si l'utilisateur lance l'exécutable compilé « code » en tapant:

```
./code a b c
```

où a, b, c sont les **paramètres du code** (des mots composés avec des lettres et des chiffres mais sans espaces dans ces mots)

Arguments fournis à l'exécution (2)

Dans le cas où le code source de la fonction principale commence par :

```
int main()
```

les paramètres a, b, c seront ignorés

Dans le cas où le code source de la fonction principale commence par :

```
int main(int argc, char **argv)
```

les paramètres a, b, c seront contenus dans argc et argv

argv est un vecteur de chaines
de caractères
argc est le nombre de chaines
dans argv

Arguments fournis à l'exécution (3)

Dans le cas où on utilise argc et argv, si l'utilisateur du binaire tape

```
./build/code test y 12
```

quand l'exécution entre dans la fonction principale, argc et argv contiendront:

```
argc : 4
argv[0] : "./build/code"
argv[1] : "test"
argv[2] : "y"
argv[3] : "12"
```

argv[0] est la chaine de caractères qui est la commande : nom du binaire y compris le chemin d'accès à ce binaire,
les autres éléments de argv sont les paramètres d'exécution.

Arguments fournis à l'exécution (3)

Si l'utilisateur du même binaire, tape

```
./build/code
```

quand l'exécution entre dans la fonction principale, argc et argv contiendront:

```
argc : 1
argv[0] : "./build/code"
```

argc est toujours au moins égal à 1

Arguments fournis à l'exécution (4)

Le programmeur doit vérifier que les paramètres fournis par l'utilisateur du code binaire sont ceux attendus (certains paramètres peuvent avoir une valeur par défaut si l'utilisateur n'en fournit pas).
Exemple de code de vérification (le code a besoin de 2 paramètres (un entier et un réel), le second a une valeur par défaut (1.5):

```
int n;
double v;
char * end;
if (argc == 1 || argc > 3) message_erreur(1);
n = strtol( argv[0], &end, 10);
if (end != NULL) message_erreur(2);
if (argc == 3) {
  v = strtod(argv[2], &end);
  if (end != NULL) message_erreur(3);
}
else
  v = 1.5;
```

`message_erreur` est une fonction (à écrire) qui affiche un message d'erreur en fonction de son paramètre

Fonctions récursives

Fonctions récursives

En C, et dans d'autres langages, une fonction peut contenir un appel à elle-même. Rappelons (voir séance 5) que chaque fois qu'on

- rentre dans une fonction, cette fonction ajoute dans la pile ses variables locales et des paramètres,
- sort d'une fonction, cette fonction enlève de la pile ses variables locales et ses paramètres.

Exemple:

```
int f(int n) {
  int k = f(n-1) * 2;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```

Ce code a pour but de calculer 2^p (p = 5, dans main.c)

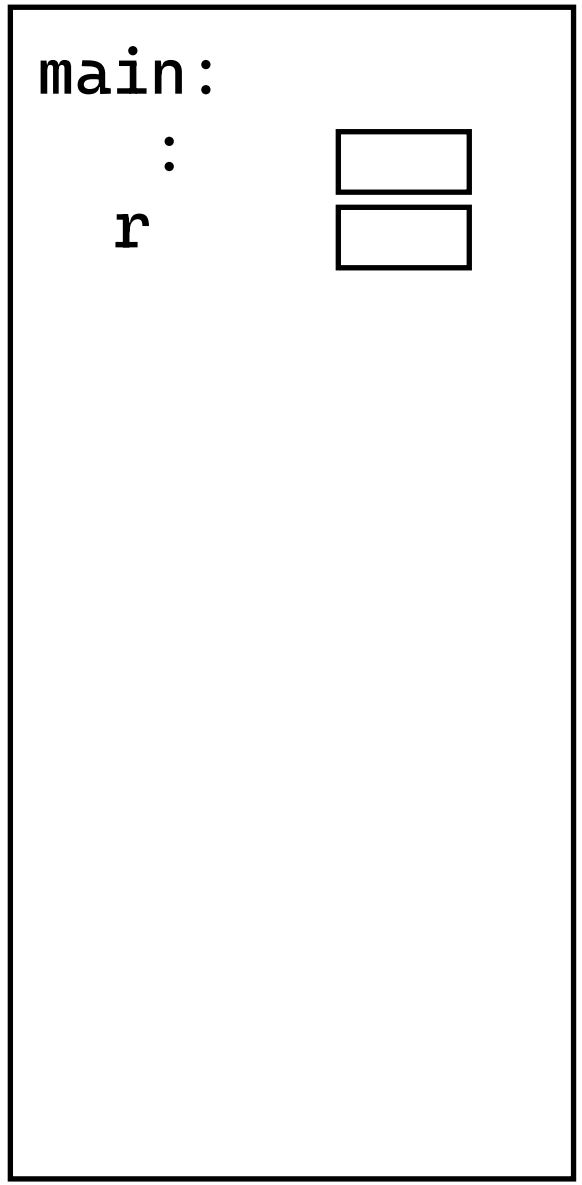
Compiler ce code :

gcc main.c –o ex1

L'exécuter … il devrait s'arrêter en affichant un message d'erreur.

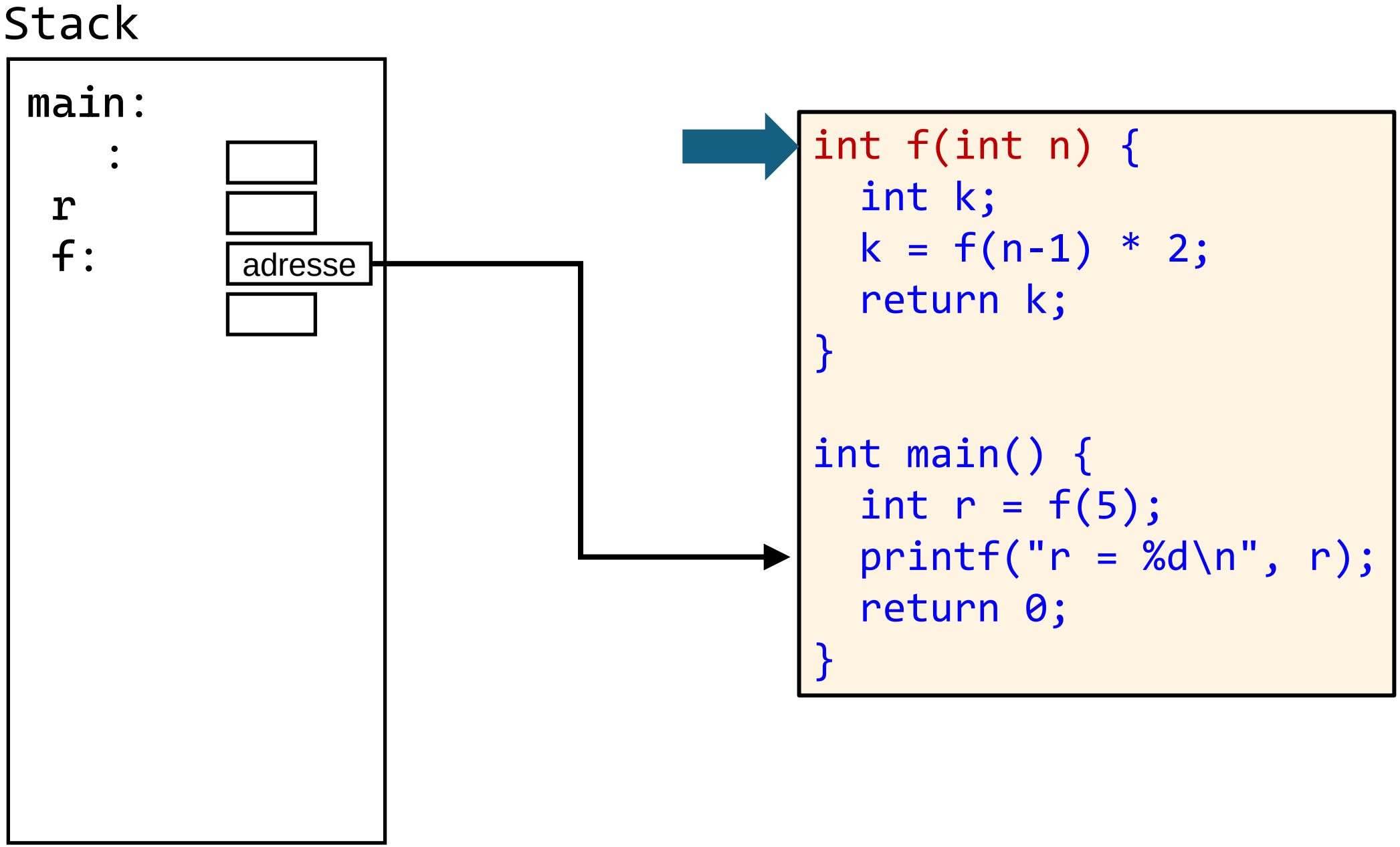
Fonctions récursives

Stack

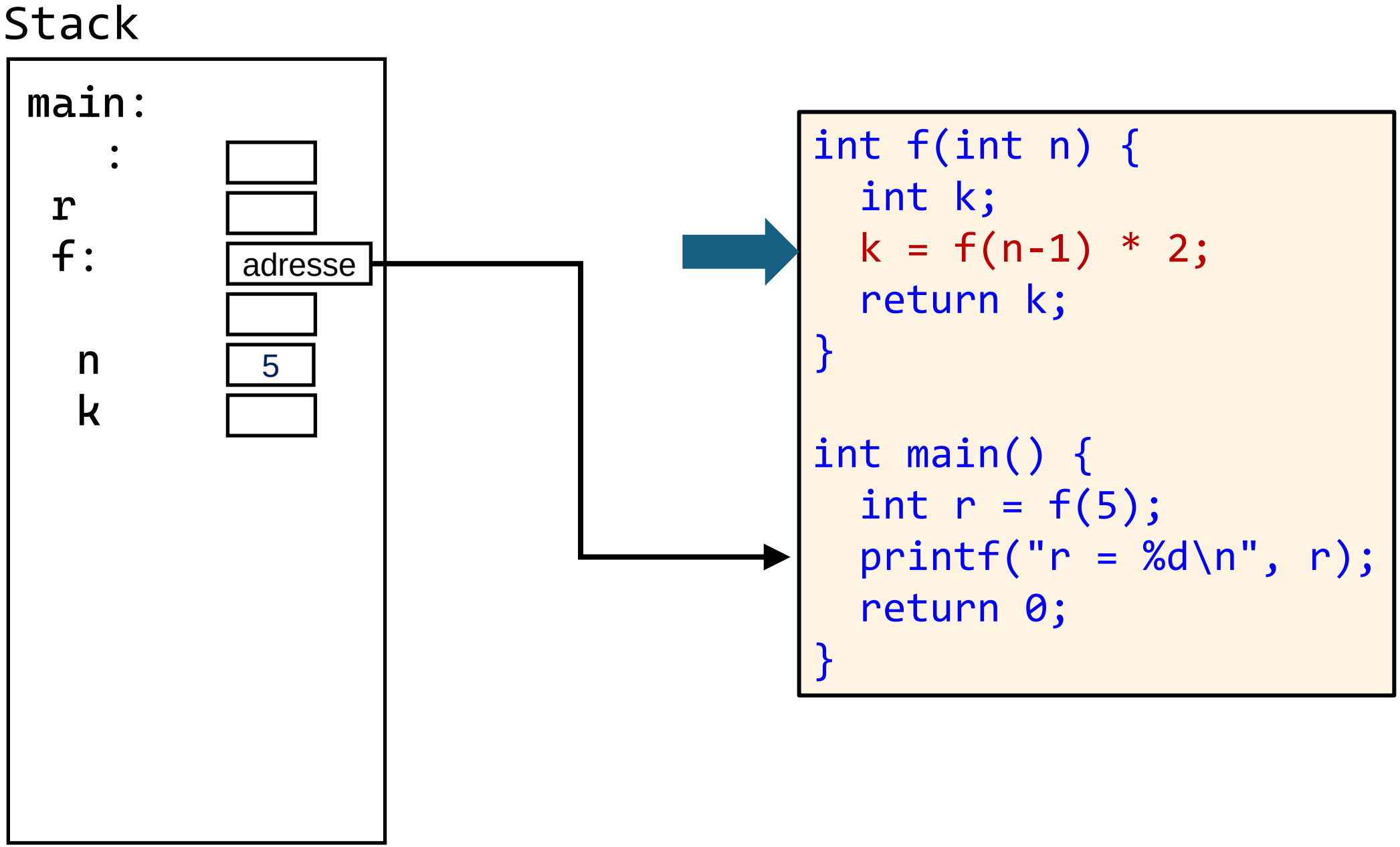



```c
int f(int n) {
  int k;
  k = f(n-1) * 2;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```

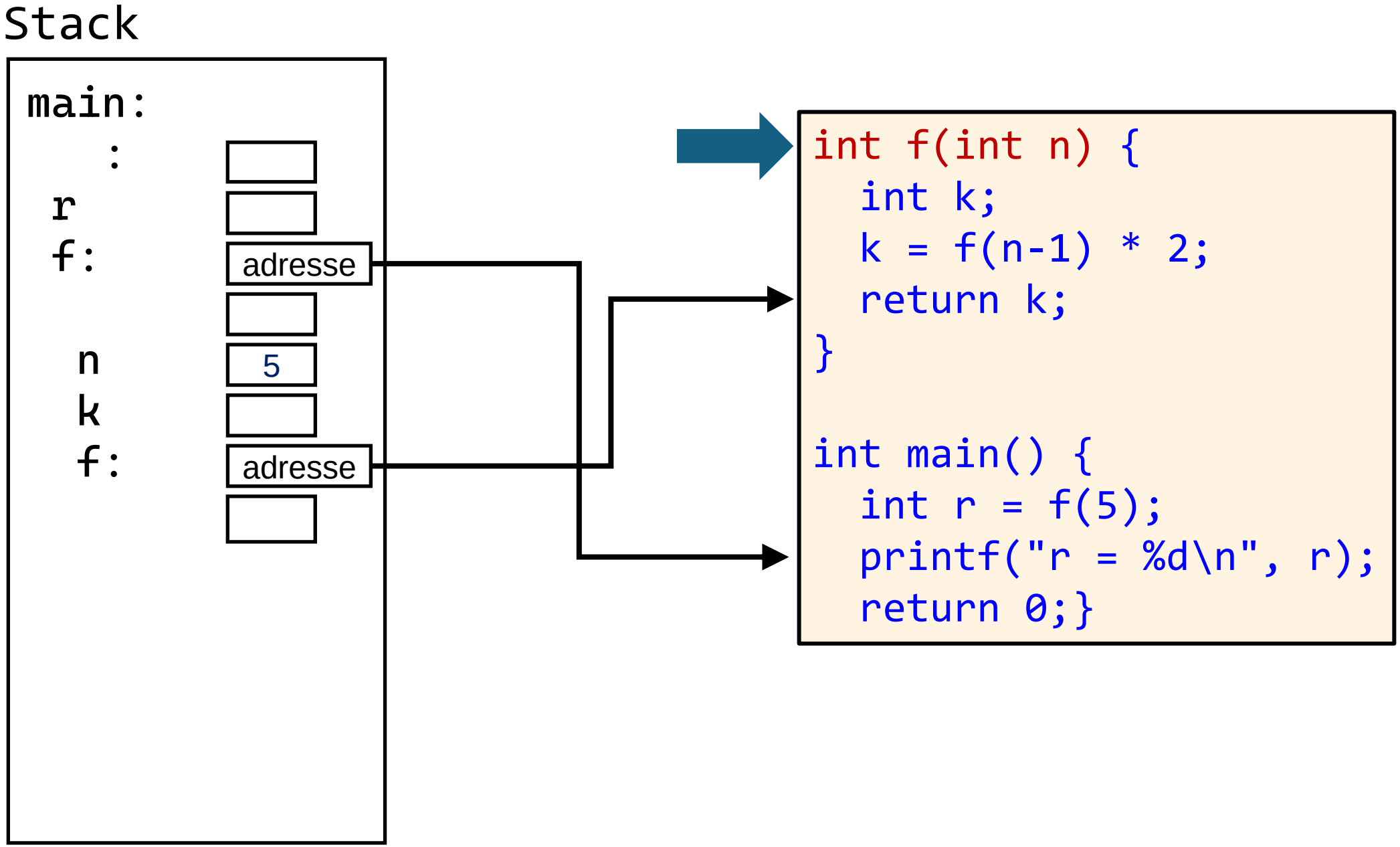
Fonctions récursives



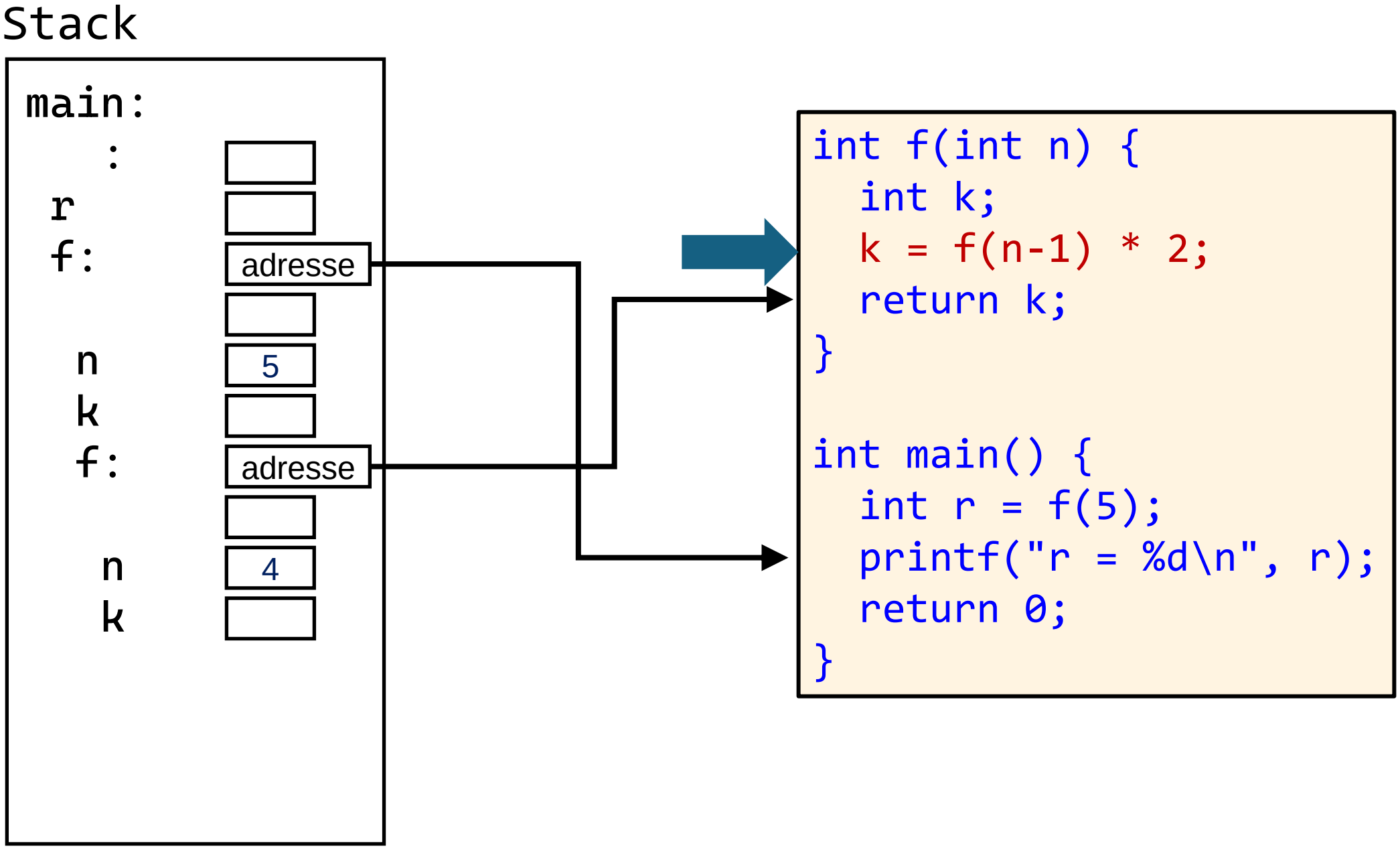
Fonctions récursives



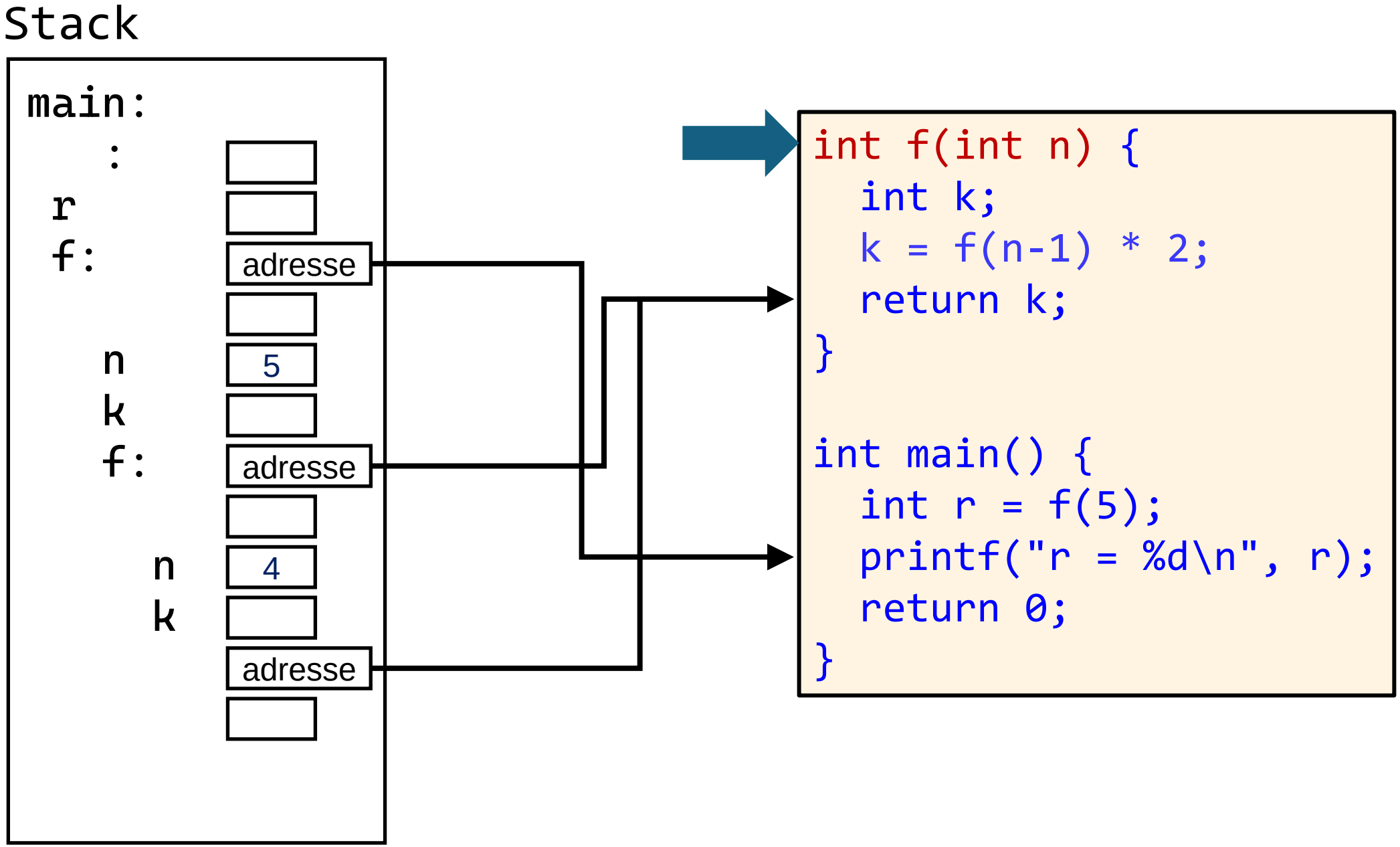
Fonctions récursives



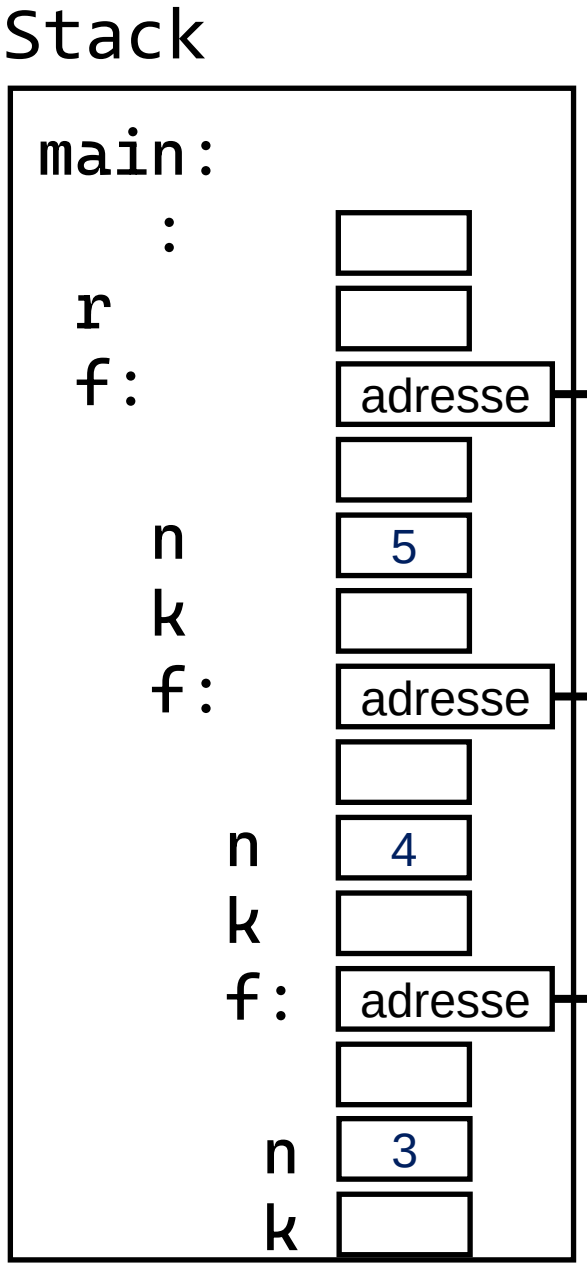
Fonctions récursives



Fonctions récursives



Fonctions récursives




```c
int f(int n) {
  int k;
  k = f(n-1) * 2;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```

Fonctions récursives

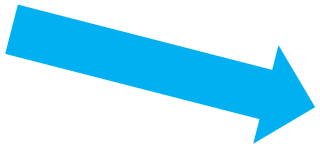
A chaque étape, les copies de la fonction f sont mises pause en attendant qu'une nouvelle copie de f s'exécute. Et cela ajoute de nouvelles variables dans la stack (pile).
L'erreur d'exécution se produit quand la stack est pleine. C'est toujours le cas ici, puisque l'algorithme n'a pas de limites sur le nombre de copies de f créées.

```
int f(int n) {
  int k = f(n-1) * 2;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```

Fonctions récursives

Il faut donc toujours mettre une limite à la récursivité (le nombre de fonctions f appelées), par exemple :

```c
int f(int n) {
  int k = f(n-1) * 2;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```



```c
int f(int n) {
  int k;
  if (n > 0) k = f(n-1) * 2;
  else       k = 1;
  return k;
}

int main() {
  int r = f(5);
  printf("r = %d\n", r);
  return 0;
}
```

Ici la limite, c'est la valeur de n telle que f(n) est explicitement connue ($f(0) = 2^0 = 1$).

Compiler ce code et l'exécuter.

Fonctions récursives (2)

2ème exemple (très connu) : calcul de la suite de Fibonacci f_k (k=0, …) où

$f_k = f_{k-1} + f_{k-2}$ (k >= 2)

$f_0 = 0$

$f_1 = 1$

La définition de la suite est récursive, on peut essayer de la calculer avec une fonction C récursive.

Examiner le code dans le répertoire fonctions_recursive/fibonacci

Compiler le code par cmake

```
cmake -S src -B build:
make -C build
```

et exécuter le code en tapant:

```
./build/fibonacci 40
```

Le temps de calcul des valeurs de f(n) pour n > 40 est de plus en plus important. Expliquer pourquoi.

Fonctions récursives (3)

Le code source contient deux autres algorithmes de calcul de la suite de Fibonacci:

- Un algorithme itératif (non récursif)
- Un algorithme récursif mais qui enregistre les valeurs déjà calculées pour ne pas recalculer plusieurs fois le même élément (programmation dynamique)

Pour utiliser ces algorithmes, tapez l'une des 2 commandes :

```
./build/fibonacci 40 iteratif
./build/fibonacci 40 dynamique
```

Vérifier que les résultats sont les mêmes avec les 3 algorithmes.

Fonctions récursives (4)

Le temps de calcul des valeurs de f(n) pour n > 40 est de plus en plus important. Expliquer pourquoi.

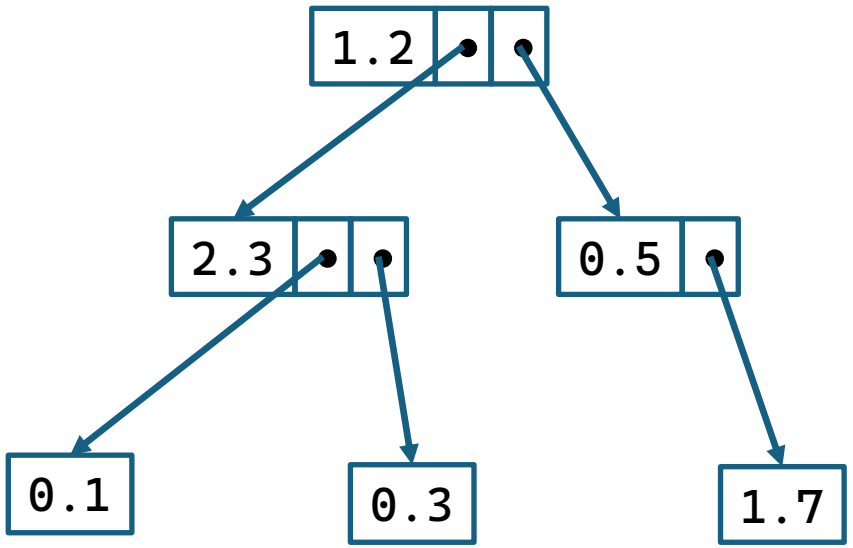
Pour certains algorithmes récursifs, il existe un algorithme non récursif équivalent, qui est souvent plus efficace (plus rapide et utilise moins de mémoire).

Dans d’autres cas, l’algorithme récursif est le seul disponible.

Structure d'arbre

Un arbre est une structure (informatique, plus générale qu'une liste) composée d'un ensemble de nœuds (avec un nœud de départ: nœud racine, des nœuds terminaux : nœuds feuilles et des nœuds intermédiaires). Les nœuds sont reliés par des connexions (sans chemin circulaire)

Exemple (chaque nœud contient un réel) :



Type nœud :

```
typedef struct _noeud
{
  double valeur;
  int nEnfants;
  struct _noeud ** enfants;
} nœud;
```

Type arbre :

```
typedef noeud * arbre;
```

Parcours récursif d’arbre

Parcourir un arbre, c’est accéder au contenu de tous les nœuds.
A ma connaissance, les seuls algorithmes qui existent sont des algorithmes récursifs.
Concevoir à titre d’exercice une fonction qui calcule la valeur maximale dans les nœuds.

Pointeurs de fonction

Pointeurs de fonctions

En C, on peut définir et utiliser des pointeurs vers le code (binaire) d'une fonction.

Par exemple, si f est une fonction à un paramètre réel qui a un résultat réel:

```
double f(double x) {
  return x*2.0;
}
```

Un pointeur pf sur f est défini et s'utilise comme suit:

```
double (*pf)(double);
pf = &f; /* ou pf = f */

X = pf(2.3) + pf(1.2);
```

Pointeurs de fonctions

Faire attention à la syntaxe (place du caractère * et des parenthèses) !!

Quelles sont les significations de 3 lignes ci-dessous ?

```
double pf(double);
```

```
double * pf(double);
```

```
double (*pf)(double);
```

Pointeurs de fonctions (2)

Exemple 1 : évaluation d'une fonction au choix (voir le répertoire pointeur_fonction/exemple1)

```
double (*p) (double);
switch (k) {
    case 1:
      p = &sin;
      break;
    case 2:
      p = &cos;
      break;
    case 3:
      p = &fois2;
      break;
  }
  x = p(3.14159);
```

Suivant la valeur de l'entier k, p(3.14…) évalue soit une fonction mathématique système sin ou cos, soit une fonction codée par l'utilisateur

Pointeurs de fonctions (3)

Exemple 2 : dérivée approchée

```
double derivee(double (*f) (double),
               double x, double dx)
{
  return (f(x+dx) - f(x-dx))/(2*dx);
}
```

Que l'on peut utiliser comme:

```
x = 1.0;
h = 0.001;
printf("x = %10.5g\n", x);
printf("  sin(x) = %12.7g\n", sin(x));
printf("  dsin/dx(x) = %12.7g\n", derivee(sin, x, h));
```

Pointeurs de fonctions (3)

Examiner le code source dans pointeur_fonction/exemple2 et compiler en tapant :

```
gcc main.c derivee.c -o ex2 -lm
```

Ne pas oublier –lm qui ajoute au code les fonctions mathématiques standards (sin, cos, …)

Une variante de cet exemple se trouve dans pointeur_fonction/exemple3 où des pointeurs de fonctions sont mis dans une variable de type struct (illustration d’un fonctionnement pré-C++)

Librairies statiques et dynamiques

Librairies

En C (et dans d'autres langages de programmation), les librairies sont un moyen de partager/réutiliser des fonctions compilées entre plusieurs codes.

Un exemple est la librairie mathématique C (contenant sin, cos, tan, exp, …) qui est utile à beaucoup de codes développés par des programmeurs. Cette librairie est fournie par tous les compilateurs.

Mais un programmeur peut aussi compiler du code source sous forme de librairie s'il estime que ce code source est utile dans plusieurs développements.

Les librairies sont du code binaire incomplet et contiennent:

- des fonctions C (mais pas la fonction principale main)
- des variables globales

Librairies statiques et dynamiques

Il existe deux sortes de librairies : statiques et dynamiques

Les librairies statiques se terminent par .a (linux, macOs), .lib (windows), les librairies dynamiques par .so (linux), .dynlib (macOs), .dll (windows)

- Les librairies statiques sont intégrées dans le code binaire final (les compilateurs font le tri des fonctions et ne gardent que les fonctions réellement utilisées).

- Les librairies dynamiques restent à l'extérieur de code (à la compilation le compilateur vérifie que tout ce qui est nécessaire se trouve dans le code source ou dans les libraires).
  Chaque fois que l'exécutable est utilisé, la machine charge les librairies dynamiques.

Avantage/inconvénients des librairies statiques

Avantages des librairies statiques :

- le code final contient tout ce qu'il faut pour l'exécuter,
- le code a les mêmes performances que le code qui n'utilise pas de librairies.

Inconvénients des librairies statiques :

- le code est plus volumineux,
- si plusieurs codes utilisent la même librairie, le binaire de la librairie existe en plusieurs copies sur les disques (dans chaque code).

Avantage/inconvénients des librairies dynamiques

Avantages des librairies dynamiques :

- le binaire commun entre plusieurs codes existe dans un seul fichier (xxx.so dans linux, xxx.dll dans windows) et le code exécutable est plus petit,
- Si on fait des corrections dans le code source d'une librairie dynamique (sans changer le nom des fonctions ou leurs paramètres), il n'est pas nécessaire de recompiler le code complet.

Inconvénients des librairies dynamiques :

- le code binaire est (un petit peu) moins rapide (parce qu'on passe toujours par des pointeurs de fonction),
- si on diffuse le code, il faut que les librairies dynamiques soient disponibles sur toutes les machines où le code est utilisé,

Exemple d’utilisation des librairies

Dans le répertoire librairies/exemple1 se trouve un exemple de construction

- d’une librairie en version statique contenant des fonctions de manipulation de vecteurs et matrices
- d’une librairie en version dynamique contenant les mêmes fonctions
- d’un code binaire exécutable qui utilise la librairie statique
- d’un code binaire exécutable qui utilise la librairie dynamique

Compilation de la librairie statique et du code qui l'utilise

Tapez les commandes pour compiler la librairie statique alglin.a:

```
gcc -I src/alglin -c src/alglin/matrice.c
gcc -I src/alglin -c src/alglin/vecteur.c
gcc -I src/alglin -c src/alglin/prodmat.c
ar r alglin.a vecteur.o matrice.o
```

Puis les commandes qui créent l'exécutable ex1_statique:

```
gcc -I src/alglin -c src/main.c
gcc main.o alglin.a -o ex1_static
```

Tester en exécutant le code:

```
./ex1_static
```

Compilation de la librairie dynamique et du code qui l'utilise

Tapez les commandes pour compiler la librairie dynamique alglin.so:

```
gcc -I src/alglin -c src/alglin/matrice.c
gcc -I src/alglin -c src/alglin/vecteur.c
gcc -I src/alglin -c src/alglin/prodmat.c
gcc -shared vecteur.o matrice.o prodmat.o -o alglin.so
```

(remarquer que les 3 premières commandes sont identiques au cas statique)

Puis les commandes qui créent l'exécutable ex1_statique:

```
gcc -I src/alglin -c src/main.c
gcc main.o alglin.so -o ex1_dynamic
```

Tester en exécutant le code:

```
./ex1_dynamic
```

Le code ne devrait pas s'exécuter; voir page suivante)

Compilation de la librairie dynamique et du code qui l'utilise (2)

Si ex1_dynamique ne s'exécute pas, taper la commande:

```
ldd ex1_dynamic
```

Cette commande liste les librairies dynamiques utilisées par ex1_dynamique:

```
linux-vdso.so.1 (0x00007fffa21fb000)
alglin.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbedce69000)
/lib64/ld-linux-x86-64.so.2 (0x00007fbedd096000)
```

Le code ne trouve pas la librairie alglin.so. Pour le lui indiquer, il faut définir la variable d'environnement LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=.
```

(a la place de ., on peut mettre le chemin complet vers le fichier alglin.so)

Utilisation de cmake

Dans les répertoires src et src/alglin se trouvent des fichier CMakeLists.txt que cmake utilise pour compiler les librairies et codes. Examinez ces fichiers.

Compiler en tapant les commandes

```
cmake -S src -B build -DCMAKE_INSTALL_PREFIX=install
make -C build install
```

Qui créent le répertoire install avec les binaires.

Exécuter les 2 versions en tapant:

```
./install/ex1_static
./install/ex1_dynamic
```

Pour pouvoir exécuter la version dynamique, il faut d'abord modifier la variable LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=./install/lib
```

Taille des binaires

Afficher la taille des fichiers binaires, en tapant

```
ls –lR install
```

Et comparer les versions statique et dynamique.