

# LANGAGE C

## Objectifs de la séance

Mise en place des fonctions en C

Modularisation

# LANGUAGE C

1. Introduction

2. Implantation des fonctions

3. Modularisation

# LANGAGE C

## Introduction

### Nécessité de restructurer le code

Programmes écrits jusqu'à présent

Intégralement codé dans le programme principal (main())  
main() obligatoire

Allongement rapide des programmes

⇒ manque de lisibilité

⇒ dette technique, problème de maintenance

Cas d'un projet :

manque de lisibilité si tout le code est dans le main()

travail collaboratif

nécessité d'adopter une autre structure

# LANGAGE C

## Introduction

### Principe de la fonction

Partie de code qui peut être appelée depuis

- ⇒ le programme principal
- ⇒ une autre fonction

Principe de fonctionnement (cours : gestion de la mémoire d'un programme)

Appel de la fonction

depuis le programme principal ou depuis une autre fonction  
déclenchement de l'exécution du bloc d'instructions

Conséquences

Evaluer et stocker les valeurs des paramètres passés à la fonction  
Initialiser les variables locales  
Exécuter le corps de la fonction  
Recopier les valeurs retournées à l'emplacement réservé  
Revenir à l'emplacement de l'appel  
Utiliser la valeur de retour dans l'expression appelante

# LANGAGE C

1. Introduction

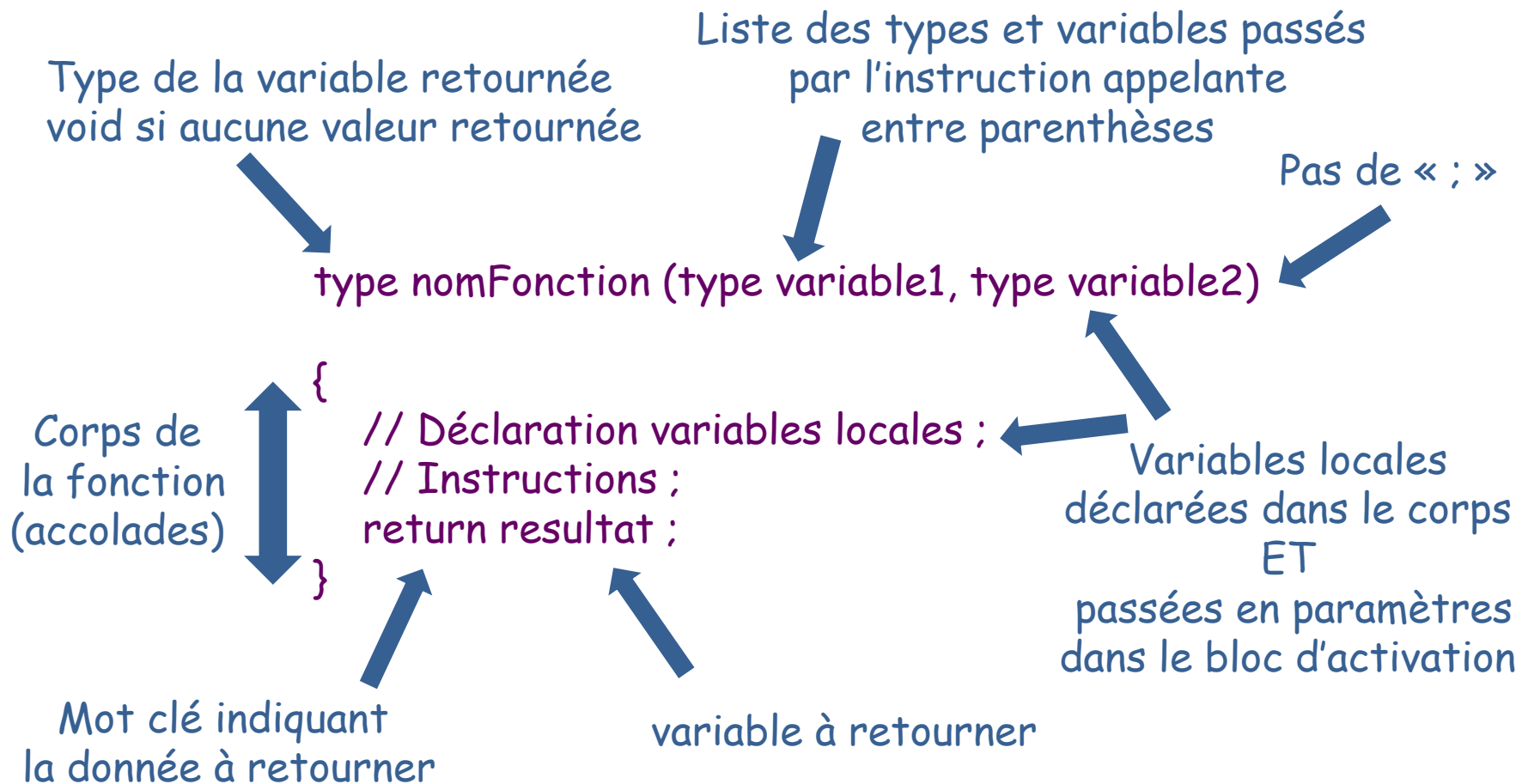
2. Implantation des fonctions

3. Modularisation

# LANGAGE C

## Implantation des fonctions

### Structure d'une fonction



# LANGAGE C

## Implantation des fonctions

Structure d'une fonction

type nomFonction (type variable1, type variable2)



Signature de la fonction  
Doit TOUJOURS être précisée  
Quel que soit le langage

(règle de bonnes pratiques de programmation)

# LANGAGE C

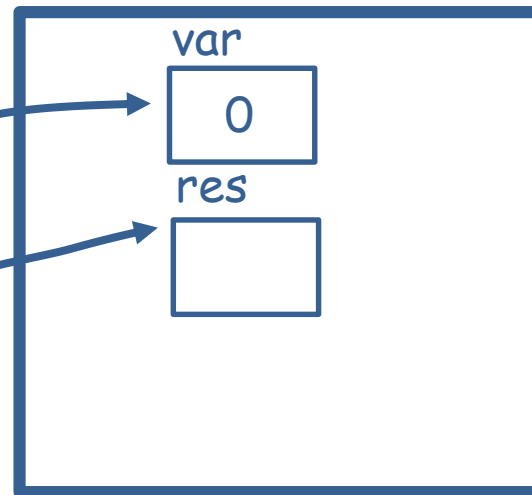
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()





# LANGAGE C

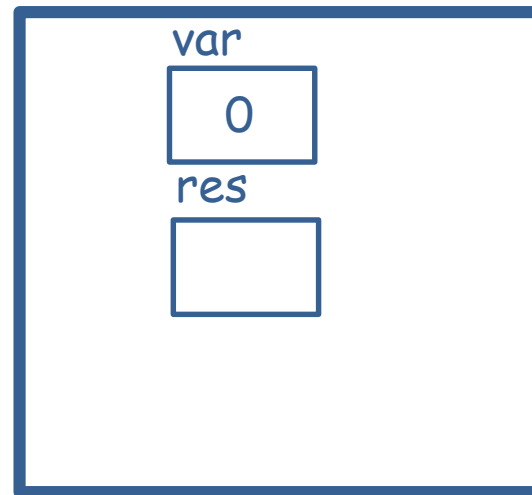
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

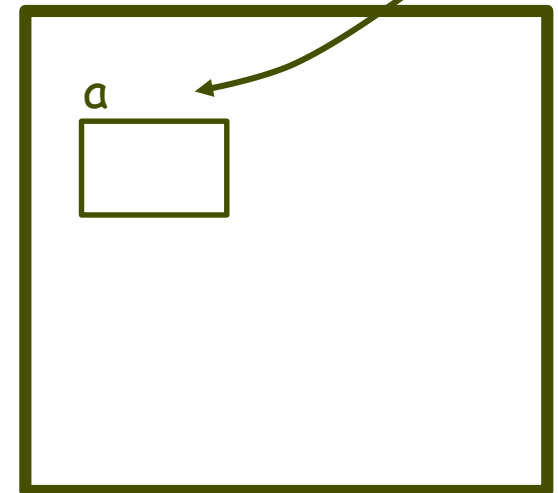
```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



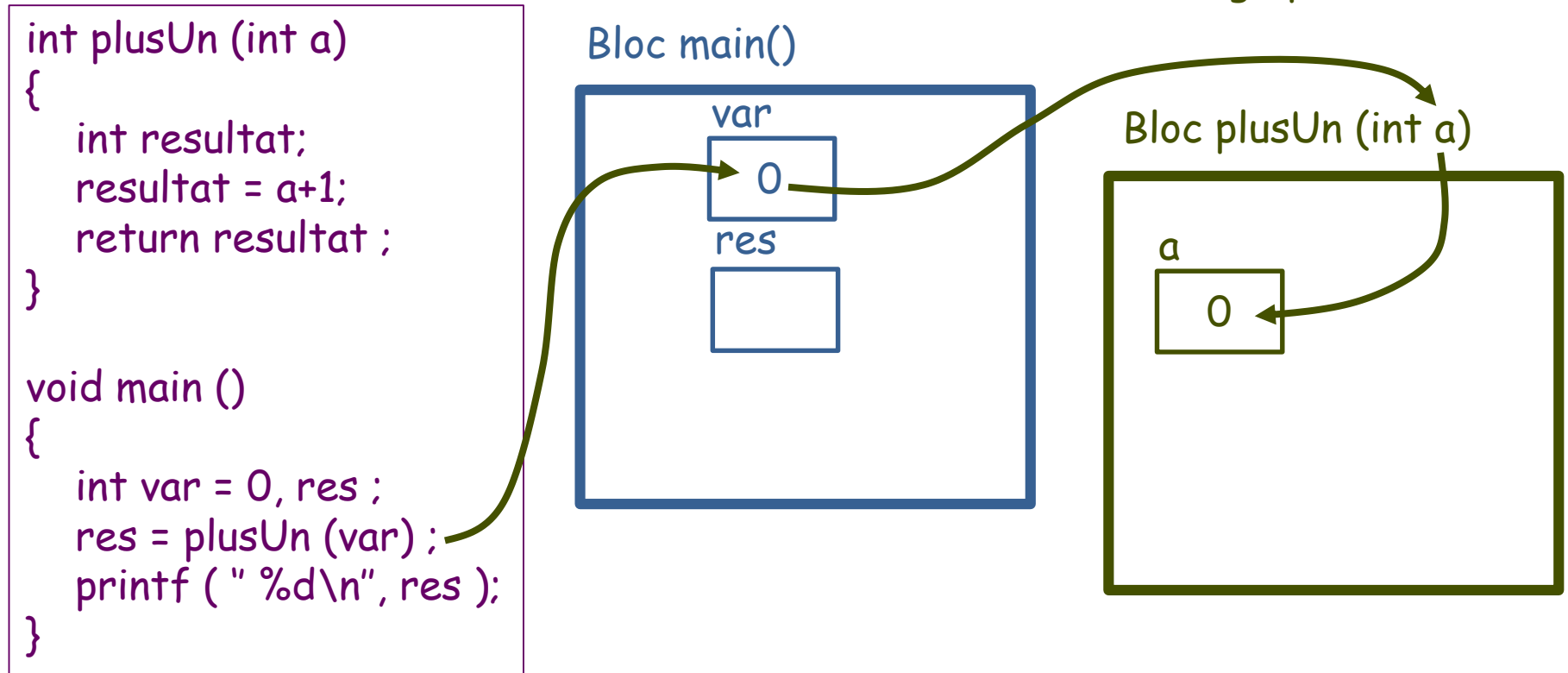
Bloc plusUn (int a)



# LANGAGE C

## Implantation des fonctions

Appel d'une fonction : déroulement du processus



# LANGAGE C

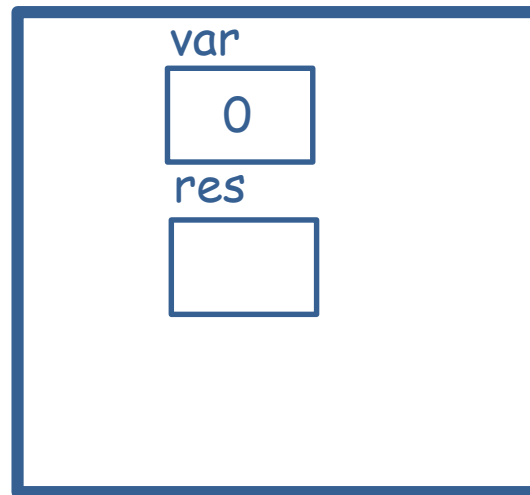
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

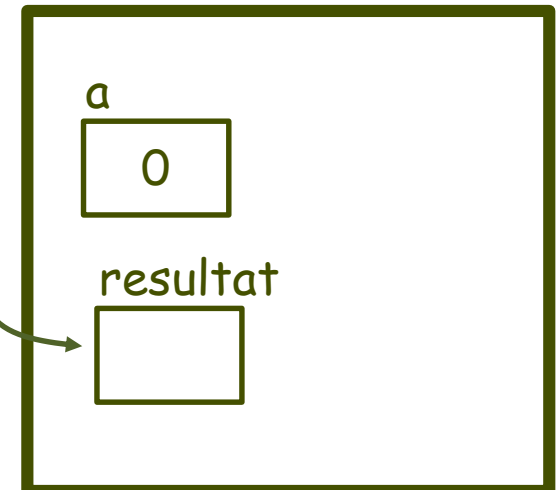
```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Bloc plusUn (int a)



# LANGAGE C

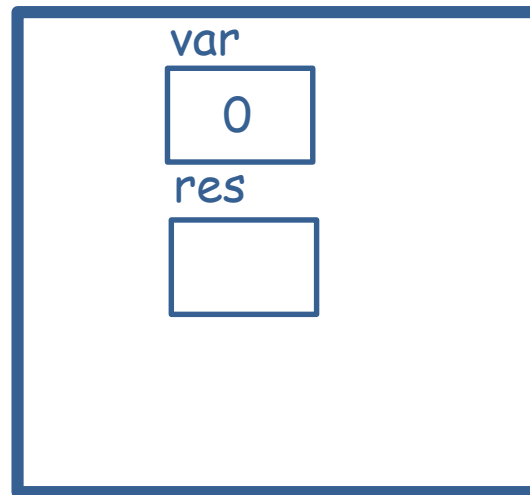
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

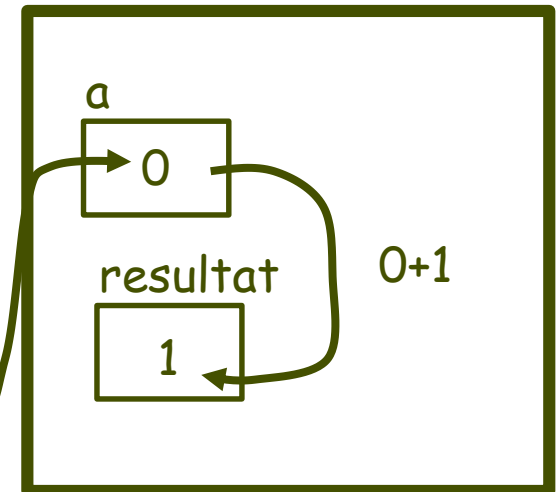
```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}

void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Bloc plusUn (int a)



# LANGAGE C

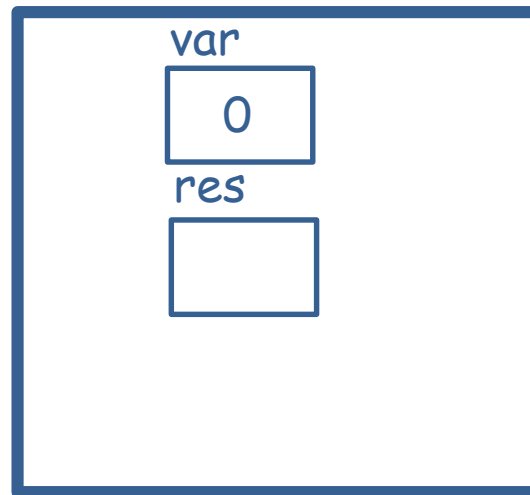
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

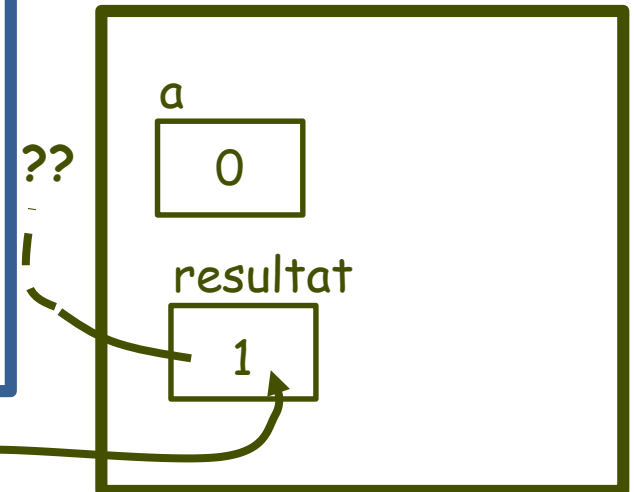
```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Bloc plusUn (int a)



# LANGAGE C

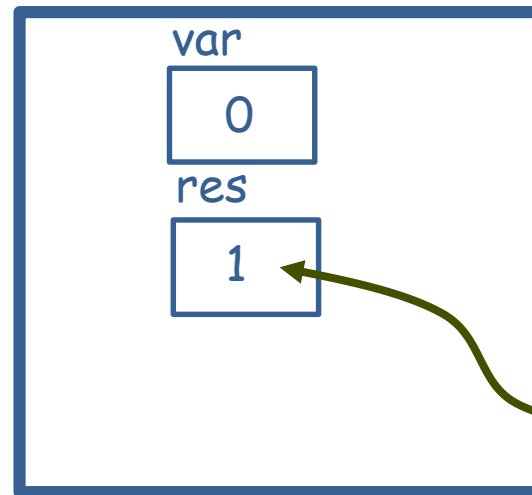
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

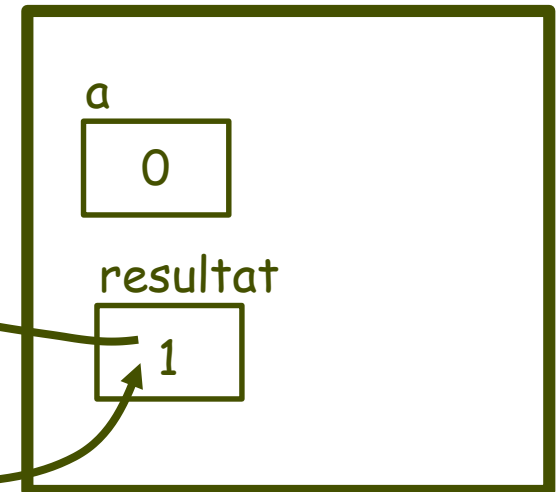
```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Bloc plusUn (int a)



# LANGAGE C

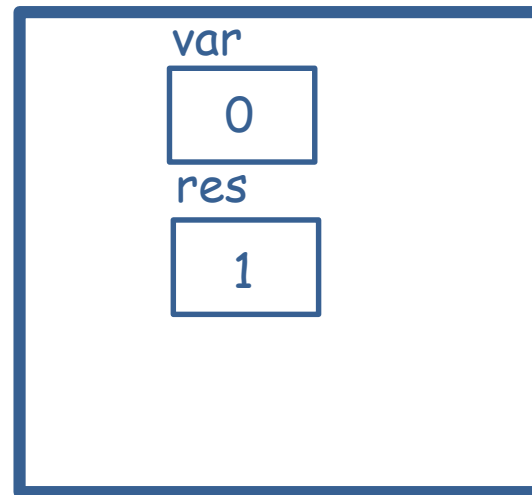
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Bloc plusUn (int a)



# LANGAGE C

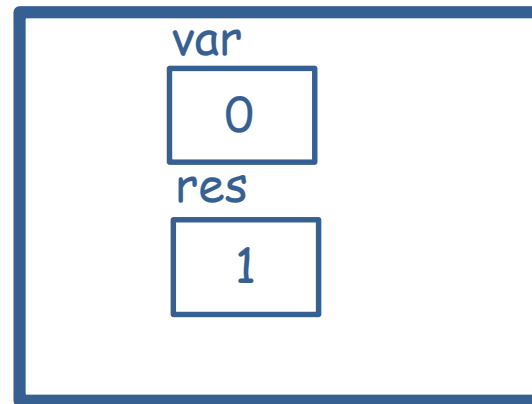
## Implantation des fonctions

Appel d'une fonction : déroulement du processus

```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}
```

```
void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```

Bloc main()



Périphérique (terminal virtuel)





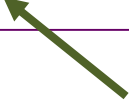
# LANGAGE C

## Implantation des fonctions

Appel d'une fonction : déroulement du processus

```
int plusUn (int a)
{
    int resultat;
    resultat = a+1;
    return resultat ;
}

void main ()
{
    int var = 0, res ;
    res = plusUn (var) ;
    printf ( " %d\n", res );
}
```



Périphérique (terminal virtuel)

1

# LANGAGE C

## Implantation des fonctions

### Signature et prototype

Compilation linéaire du code

⇒ plusieurs fonctions : écrites avant l'instruction appelante  
Sinon, erreur de compilation

Solution : déclaration des fonctions

⇒ Prototypes des fonctions entre les directives préprocesseur  
et les déclarations de types et structures personnalisés

type nomFonction (type variable1, type variable2)      signature

type nomFonction (type variable1, type variable2) ;      prototype



# LANGAGE C

## Implantation des fonctions

### Signature et prototype

```
# include <stdio.h>

Type fonction3 (type variable)
{
    ...
}

Type fonction1 (type variable)
{
    appel fonction2
}
```

```
Type fonction2 (type variable)
{
    ...
}

void main()
{
    ...
}
```

Echec de la compilation  
fonction2 non déclarée

# LANGAGE C

## Implantation des fonctions

### Signature et prototype

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

```
Type fonction1 (type variable) ;  
Type fonction2 (type variable) ;  
Type fonction3 (type variable) ;
```

```
Type fonction3 (type variable)  
{  
    ...  
}
```

```
Type fonction1 (type variable)  
{  
    ...  
}
```

```
Type fonction2 (type variable)  
{  
    ...  
}  
  
void main()  
{  
    ...  
}
```

# LANGAGE C

## Implantation des fonctions

### Passage par valeur

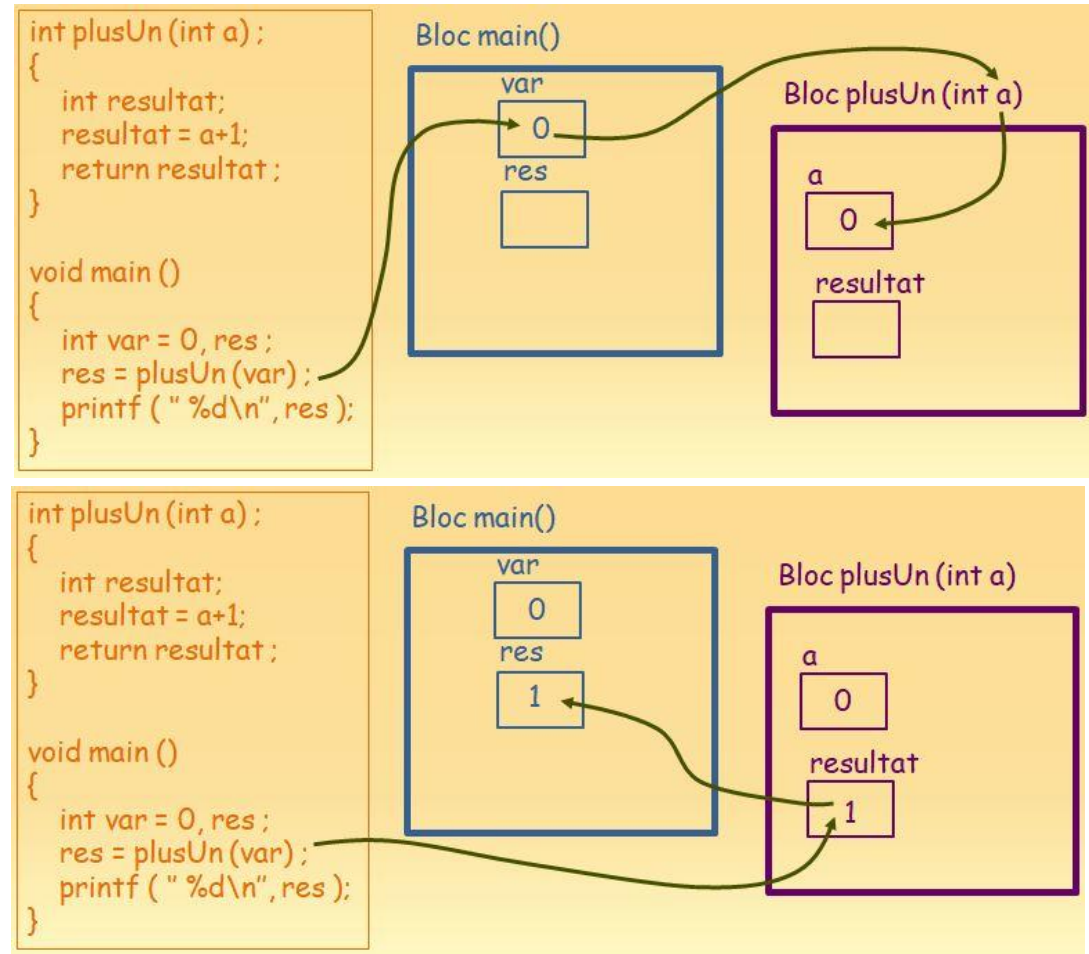
Valeur de la variable passée  
et stockée dans une variable  
locale

Variable initiale (var ou  
resultat) non modifiée

Autre possibilité :  
Passage par adresse

⇒ pointeur

⇒ modification de la  
variable initiale



# LANGAGE C

1. Introduction

2. Implantation des fonctions

3. Modularisation

# LANGAGE C

## Modularisation

### Introduction

Projet volumineux

⇒ grand nombre de fonctions

⇒ problème de lisibilité et de maintenance

Regroupement de fonctions en modules

Importation des modules dans le programme :

```
# include "module_perso.h"
```



Modules natifs : <nomModule.h >

Modules personnels : "nomModule.h"

Modules compilés séparément

# LANGAGE C

## Modularisation

### Introduction

Regroupement de fonctions en modules

Module : nomModule.c

regroupe toutes les fonctions

importe les modules utiles (compilation séparée)

définit les types utilisés

Header : nomHeader.h

regroupe les prototypes du module

Importation des modules dans le programme :

```
# include "module_perso.h"
```

⇒ attention : un module ne peut être importé qu'une fois par projet



# LANGAGE C

## Modularisation

### Mécanisme de protection

Grand nombre de modules s'utilisant les uns les autres ⇒ plusieurs headers

Modules compilés séparément

- ⇒ Déclaration multiple de fonctions (prototype) ou de variables globales
- ⇒ Risque d'erreur de compilation

Mécanisme de protection

⇒ ifndef (if not define) pour tester si le nom est présent en mémoire

⇒ Si ce n'est pas le cas :

Définir le nom de la variable avec le define  
Importer le fichier d'en-tête

⇒ Si c'est le cas, fin du test (endif)

```
# ifndef EnTete  
# define EnTete  
# include "EnTete.h"  
# endif
```

Ne pas utiliser le define  
pour déclarer les variables