Analyse et programmation 1

Les fonctions

heig-vd

Où en sommes-nous ?

Thème

- Introduction
- Aperçu du fonctionnement d'un ordinateur
- Introduction au langage C
- Représentation et traitement de l'information
  - Les types de données de base et leurs opérations
- Contrôle du déroulement d'un programme
  - Les structures de contrôle
- Décomposition d'un algorithme complexe
  - Les fonctions
- Approfondissements

heig-vd

Thèmes abordés

- Comment écrire une application complexe ?
  - Diviser pour régner.
  - Décomposer en fonctions plus simples.
- Comment créer une fonction ?
- Comment échanger de l'information avec une fonction ?
  - Passage de paramètres
- Comment tout cela fonctionne-t-il effectivement ?
  - Fonctionnement en mémoire.
- Comment décomposer efficacement un problème ?
  - Méthode de raffinage successif.
- Quelques éléments avancés
  - Les variables locales et globales.

Introduction aux fonctions

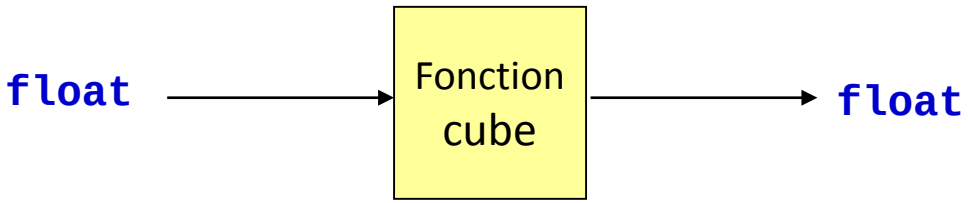
Un exemple pour commencer

- On veut disposer d'une nouvelle fonction
  - permettant de calculer le cube d'un nombre.
  - s'appelant « **cube** ».
  - que l'on pourrait utiliser sous la forme suivante:

```
x = cube(a) * 2.5;
```

- Le compilateur doit connaître cette fonction
  - Son nom, ses paramètres, le type de son résultat.
  - On lui indique en écrivant le **prototype** de la fonction :

```
float cube(float);
```



Introduction aux fonctions

Appel d'une fonction simple

```
float cube(float); // prototype de la fonction cube

int main(void)
{
    float c, a;
    a = 1.5;
    c = cube(a); // première utilisation de cube
    printf("c : %f\n", c);
    c = cube(a) * 3.0; // deuxième utilisation de cube
    printf("c : %f\n", c);
    // troisième utilisation :
    printf("cube de 2 : %f\n", cube(2.0f));
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Introduction aux fonctions

Définition d'une fonction simple

- La fonction cube
  - Est déclarée :
    - Son nom, le type de son paramètre et de son résultat sont connus.
  - N'est pas encore définie
    - Son contenu n'a pas encore été défini.
  - La compilation du programme précédent donne une erreur
    - **`unresolved external symbol "float cube(float)" referenced in function _main`**
- Nous devons donc définir le corps de la fonction
  - Donner les instructions qui constituent le corps de la fonction.
  - Ainsi le programme pourra être compilé et lié.

Introduction aux fonctions

Définition d'une fonction simple

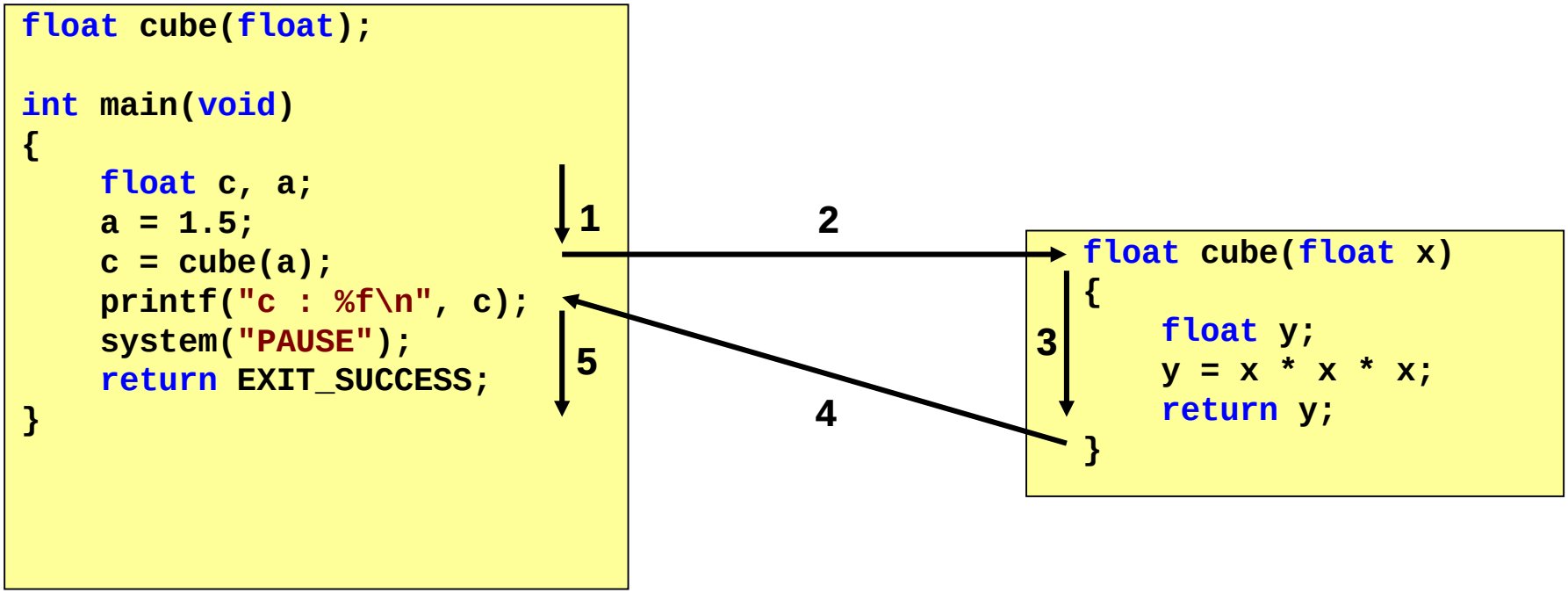
```
float cube(float); // prototype de la fonction cube → (déclaration)

int main(void)
{
    float c, a;
    a = 1.5;
    c = cube(a); // première utilisation de cube
    printf("c : %f\n", c);
    c = cube(a) * 3.0f; // deuxième utilisation de cube
    printf("c : %f\n", c);
    printf("cube de 2 : %f\n", cube(2.0f)); // troisième utilisation
    system("PAUSE");
   return EXIT_SUCCESS;
}

float cube(float x) // code de la fonction → (définition)
{
    float y;
    y = x * x * x;
    return y;
}
```

Introduction aux fonctions

Fonctionnement



Passage de paramètres

Déclaration des paramètres formels

- Paramètre formel
  - Désigne un paramètre **d'entrée de la fonction**.
  - Déclaré dans la définition de la fonction.
- Pour déclarer un paramètre formel, il faut indiquer
  - Son type
  - Son nom
- Plusieurs paramètres formels peuvent être déclarés
  - Il faut séparer les définitions par une virgule.
  - Le type doit être répété pour chaque paramètre
    - Même s'il est identique au précédent.
    - Contrairement aux déclarations de variable.

Passage de paramètres

Déclaration des paramètres formels

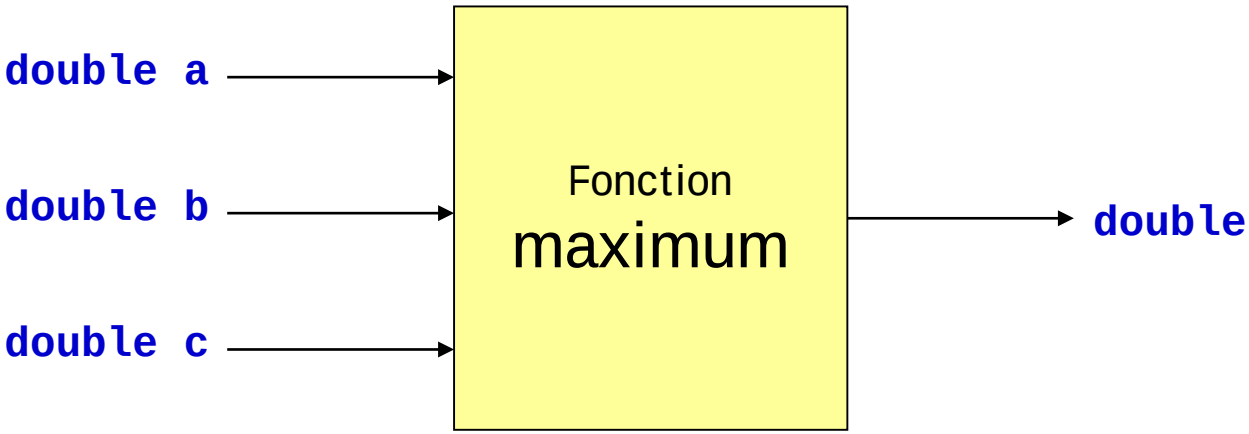
- Exemple

```
double maximum(double a, double b, double c)
```

1er paramètre formel

2nd paramètre formel

3ème paramètre formel



Passage de paramètres

Déclaration des paramètres formels

- Les paramètres formels sont utilisables
  - comme des variables locales.
  - à l'intérieur de la fonction seulement.
- Exemple

```
double maximum(double a,  double b,  double c)
{
    double resultat;
    resultat = a;
    if (b > resultat)
        resultat = b;
    if (c > resultat)
        resultat = c;
    return resultat;
}
```

1er paramètre formel

2nd paramètre formel

3ème paramètre formel

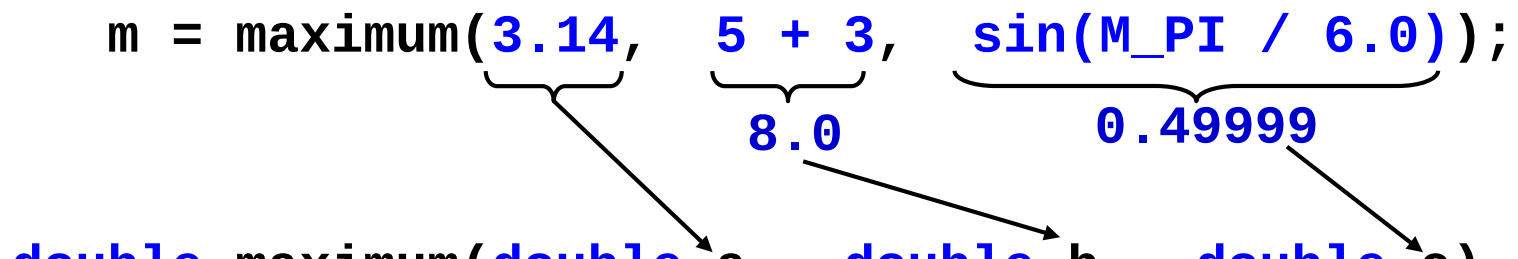
Passage de paramètres

Passage des paramètres effectifs

- **Paramètre effectif**
  - C'est la **valeur** ou **l'expression** passée à **l'appel d'une fonction**.
  - Il est identifié par sa position dans l'appel.
- Lors de l'appel d'une fonction
  - Les paramètres effectifs sont évalués.
  - Leur valeur est placée dans les paramètres formels
    - Qui sont des variables locales de la fonction appelée.
  - Ensuite, l'exécution de la fonction proprement dite démarre.

Passage de paramètres

Passage des paramètres effectifs - Illustration




```
{
    double resultat;
    resultat = a;
    if (b > resultat)
        resultat = b;
    if (c > resultat)
        resultat = c;
    return resultat;
}
```

Passage de paramètres

Ordre d'évaluation des paramètres

- Lorsqu'il y a plusieurs paramètres
  - L'ordre dans lequel ces paramètres seront évalués n'est pas défini.
- L'ordre d'évaluation dépend du compilateur.
  - Qui peut choisir un ordre différent à chaque fois.
  - Pour optimiser les performances.
  - En général, cet ordre n'a pas d'importance.
  - Sauf lorsque l'évaluation des paramètres a un effet de bord.
- Exemple avec effet de bord

```
int i = 0;
m = maximum(i++,  i,  --i); // (codage absurde)
```

heig-vd

Passage de paramètres

Les paramètres sont passés par valeur

- Les paramètres effectifs
  - Sont d'abord évalués.
  - Leur valeur est recopiée dans le paramètre formel.
  - Le paramètre formel est une variable locale.
  - Initialisée avec la valeur du paramètre formel lors de l'appel de la fonction.
- Ce mode de passage de paramètre a un nom
  - Il s'appelle le «passage de paramètre par valeur » ou "paramètre d'entrée".
  - Seule la valeur d'un paramètre est passée à la fonction appelée.
  - Il se retrouve dans la plupart des langages de programmation.
- Avec le langage C
  - C'est le seul mode de passage de paramètre.
  - Il est cependant possible d'imiter le passage par variable (par référence).

Passage de paramètres

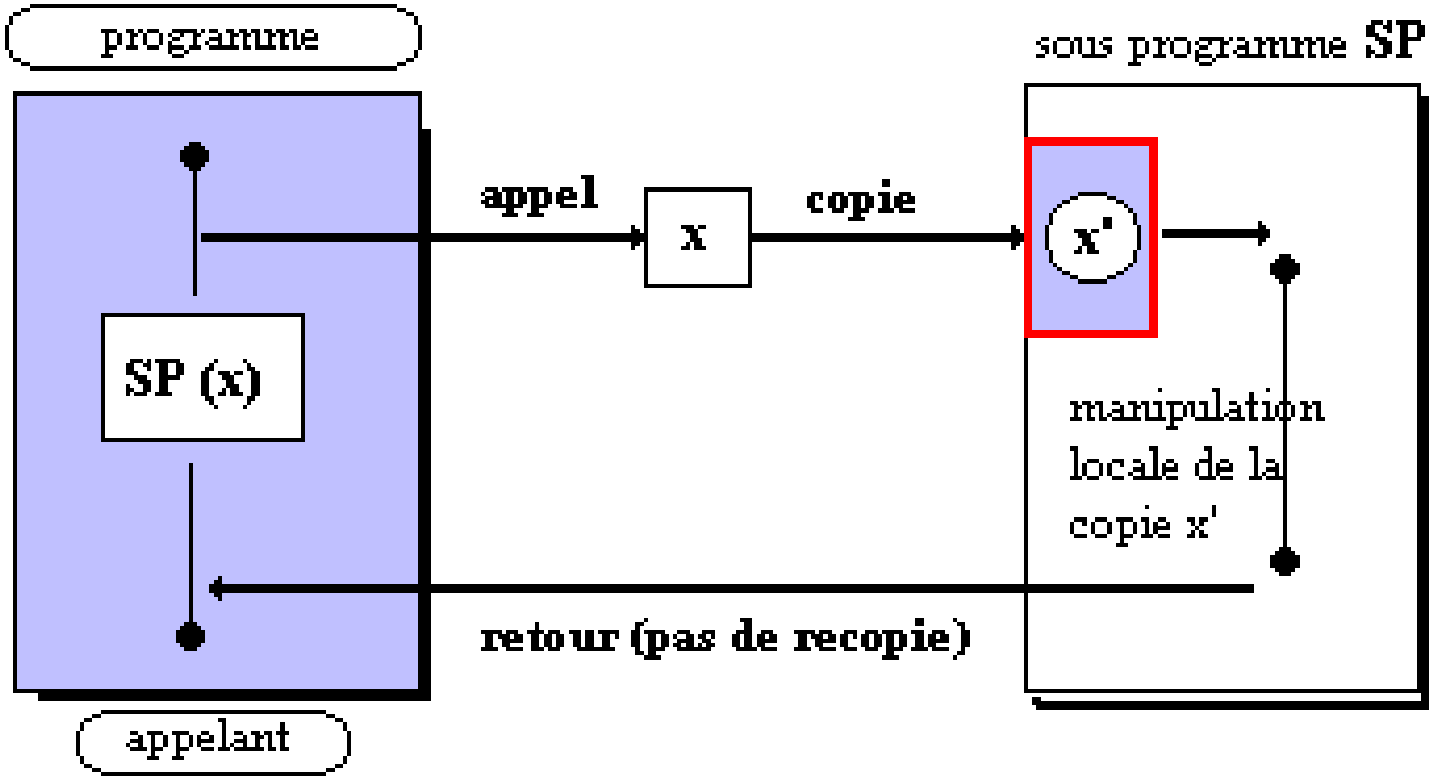
Affectation des paramètres formels dans la fonction

```c
int fonction_essai(int parametre)
{    // modifie parametre, mais pas i
     parametre = parametre + 1;
     return parametre;
}

int main(void)
{
     int i;
     i = 5;
     fonction_essai(i);
     printf("%d\n", i);        // affiche 5
     i = fonction_essai(i);
     printf("%d\n", i);        // affiche 6
    return EXIT_SUCCESS;
}
```

Passage de paramètres

Affectation des paramètres formels dans la fonction



Passage de paramètres

Affectation des paramètres formels dans la fonction

- L'affectation des paramètres formels dans la fonction
  - Elle est sans effet sur une variable passée en paramètre effectif.
  - En effet, la valeur de la variable passée en paramètre effectif est recopiée dans le paramètre formel
  - Donc, la fonction ne travaille pas directement avec la variable passée en paramètre.
- C'est le principe même du passage par valeur.
- Certains langages supportent un autre mode de passage
  - Appelé le passage « par variable » ou « par référence»
  - Lorsqu'il est utilisé, les variables passées en paramètre peuvent être directement modifiées par la fonction appelée.

Passage de paramètres

Fonctions sans paramètres

- Déclaration
  - Il existe 2 formes de déclaration pour les fonctions sans paramètres
  - Première forme (ne plus utiliser):

```
double lire_capteur_temperature(); // ancienne forme
```

  - Deuxième forme :

```
double lire_capteur_temperature(void); //nouvelle forme
```

- Appel
  - Même s'il n'y a pas de paramètres, **il faut placer des parenthèses**.

```
double temperature;
temperature = lire_capteur_temperature();
```

Fonction avec résultat

Retourner un résultat

- **Pour retourner un résultat**
  - Une fonction utilise l'instruction **`return`**
  - La valeur passée correspond au résultat de la fonction.
  - Il n'y a pas besoin de parenthèses, mais elles sont tolérées
    `return (1)`.
- Deuxième effet
  - Lorsqu'une fonction utilise return, son exécution se termine.
- Exemple

```
int fonction(float x)
{
    return x + 1;
    printf("cette ligne n'est jamais executee\n");
}
```

Fonction avec résultat

Comportement en cas d'absence du return

- Si l'instruction return n'est pas exécutée dans une fonction
  - La valeur de retour est quelconque.
  - **C'est un comportement à éviter absolument.**
  - De nombreux compilateurs le signalent tout en le tolérant.
- Exemple

```
double carre(double x)
{
    double resultat;
    resultat = x * x;
    // il manque le return !
}

int main(void)
{
    double f;
    f = carre(2.0);
    printf("%f\n", f); // valeur quelconque affichée
    return EXIT_SUCCESS;
}
```

Fonction avec résultat

Utiliser une fonction qui retourne un résultat

- Une fonction retournant un résultat
  - Peut être utilisée comme n'importe quelle fonction mathématique prédéfinie.
  - Peut être intégrée dans des expressions arithmétiques.
  - On peut aussi ignorer le résultat. Voir printf si dessous.

- Exemple

```
double carre(double x)
{
    double resultat;
    resultat = x * x;
    return resultat;
}

int main(void)
{
    double f;
    f = 2.0 * carre(9.0) + 5.0;
    printf("%f\n", f); // le résultat de printf est ignoré
    return EXIT_SUCCESS;
}
```

Fonction avec résultat

Déclarer une fonction qui ne renvoie pas de résultat

- Découpage d'un programme en sous fonctions
  - On peut utiliser les fonctions comme unité de programmation pour découper un programme en sous programmes.
  - Exemple
    - Application complexe : « main » contient un menu principal.
    - Ce menu principal permet d'accéder à des sous menus.
    - On crée des fonctions pour gérer chaque sous menu.
    - Ces fonctions gèrent leur menu de façon autonome.
    - Elles n'ont pas de résultat à retourner.
- Créer une fonction ne retournant pas de résultat
  - Déclarer le type de retour **`void`** (signifie vide), à partir de C89

```
void sous_menu1(void)
{
}
```

Fonction avec résultat

return dans une fonction void

- Une fonction void ne renvoie pas de résultat.
- On peut cependant utiliser l'instruction return.
  - Elle sert alors simplement à terminer l'exécution de la fonction.
  - syntaxe : `return;`

Prototype de fonction

Rôle

- Que se passe-t-il s'il manque le prototype ?

```
int main(void)
{
    double f;
    f = carre(2.0);
    printf("%f\n", f);
    return EXIT_SUCCESS;
}

double carre(double x)
{
    double resultat;
    resultat = x * x;
    return resultat;
}
```

Première occurrence de carre.
En l'absence de prototype, le compilateur suppose
`int carre(int)`

Deuxième occurrence de carre.
Le compilateur voit qu'il a deux définitions différentes de la fonction carre → **Il génère une erreur de compilation.**

Prototype de fonction

Rôle

- Que se passe t il s'il manque le prototype ?
  - Lorsque le compilateur voit une nouvelle fonction, il déduit sa déclaration du nombre de paramètres.
  - **Il donne un type int à chaque paramètre et un résultat int à la fonction.**
  - En compilant la définition de la fonction, il repère un conflit
    - Génère un message d'erreur explicite.
- Avec la compilation séparée
  - Il est possible de placer la définition d'une fonction dans un fichier séparé (expliqué dans INFO2).
  - Dans ce cas, le compilateur ne remarque aucun problème.
  - A l'exécution, il y a discordance des types de données
    - Dysfonctionnement du programme assuré !
    - Souvent difficile à localiser !

Prototype de fonction

Rôle

- Déclaration du prototype de la fonction

```
double carre(double x);

int main(void)
{
    double f;
    f = carre(2.0);
    printf("%f\n", f);
    return EXIT_SUCCESS;
}

double carre(double x)
{
    double resultat;
    resultat = x * x;
    return resultat;
}
```

Le compilateur prend connaissance du prototype de la fonction carre.

Lorsque le compilateur rencontre un appel à cette fonction, il dispose de toute l'information nécessaire sur les paramètres et le résultat.
Donc, ça fonctionne.

Définition de la fonction carre.
Elle concorde avec la déclaration du prototype, donc ça fonctionne bien.

Prototype de fonction

Rôle

- Autre solution :
  - En plaçant la définition de la fonction avant son utilisation

```
double carre(double x)
{
    double resultat;
    resultat = x * x;
    return resultat;
}

int main(void)
{
    double f;
    f = carre(2.0);
    printf("%f\n", f);
    return EXIT_SUCCESS;
}
```

En compilant la définition de la fonction, le compilateur prend connaissance de son prototype.

Lorsque le compilateur rencontre un appel à cette fonction, il dispose de toute l'information nécessaire sur les paramètres et le résultat.
Donc, ça fonctionne.

Prototype de fonction

Rôle

- Le prototype de fonction
  - Indique au compilateur le type des paramètres et du résultat d'une fonction.
  - Evite que le compilateur ne fasse des suppositions fausses.
- Syntaxe
  - Il faut indiquer le type des paramètres.
  - **Le nom des paramètres est fortement conseillé**, mais pas requis.
  - Il est souvent utile de le répéter, car il explique le sens du paramètre à la personne qui relit le code.
- Exemples valides

```
double calculer_courant(double, double);  // Déconseillé
double calculer_courant(double resistance, double tension);
```

Prototype de fonction
Rôle

- Emplacement
  - Le prototype d'une fonction doit être déclaré avant le premier appel à cette fonction.
  - Normalement, il doit être déclaré en dehors de toute fonction.
    - Certains compilateurs acceptent qu'il soit déclaré dans une fonction.
- Usage recommandé
  - Le comportement "**par défaut**" du compilateur est source d'ennui.
    - Un résultat et des paramètres int ne correspondent pas au cas général.
  - **Donc, toujours déclarer un prototype avant d'utiliser une fonction.**
  - Même pour les fonctions int f(int).

heig-vd

Prototype de fonction

Contenu des fichiers « include »

- Fonctions standards utilisées dans les programmes
  - Nous avons déjà utilisé printf, scanf, sin, …
- Pour que le compilateur connaisse leur prototype

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

- Que trouve-t-on dans ces fichiers
  - Une liste de prototypes de fonctions.
  - Des déclarations de constantes.
  - Exemple extrait de math.h

```
. . .
double  __cdecl sin(__in double _X);
double  __cdecl sinh(__in double _X);
double  __cdecl tan(__in double _X);
. . .
```

Complément

Les anciennes formes de déclaration de fonction

- Définie par la première norme du C, appelée K & R
- Les prototypes de fonction avaient la forme suivante

```
double carre(x) // paramètres seulement nommés
    double x; // type des paramètres précisé ici
{
    double resultat;
    resultat = x * x;
    return resultat;
}
```

- Pas acceptée par les compilateurs C++ et C99.
- **A éviter absolument.**
- Se trouve encore dans certaines bibliothèques.


Exercices
Algorithmes et
fonctions

Les fonctions

Les paramètres et les variables locales

- Utiliser des variables pour réaliser un traitement
  - Les variables locales.
  - Visibilité, durée d'existence.
- Enchaînement de fonctions
- Comment tout cela fonctionne-t-il en mémoire ?
  - La pile
- Les fonctions avec un nombre variable de paramètres
  - Comment en créer ?

Variables locales

Quelle utilité

- Les fonctions utilisent l'information reçue en paramètres.

- Pour réaliser le traitement attendu, elles ont besoin
  - de calculer des résultats intermédiaires : variables temporaires.
  - de réaliser des boucles for : variables de boucles.

- Il est donc indispensable de pouvoir créer des variables dans les fonctions.

Variables locales

Déclaration et initialisation

- Notion de variable locale
  - Les variables déclarées au début d'un bloc sont dites locales.
  - Elles existent à partir de leur déclaration jusqu'à la fin du bloc.
  - Elles peuvent être initialisées au moment de leur déclaration.
- Exemple

```
void afficher_table_multiplication(long valeur)
{
    long i; // variable locale, déclarée ici
    printf("Table de multiplication de %ld\n", valeur);
    for (i = 1; i <= 10; i++)
        printf("%ld * %ld = %ld\n", i, valeur, i * valeur);
    printf("\n");
}   // après cette ligne, i n'est plus visible
```

Variables locales

Visibilité

- Une variable locale est « visible » par le programme
  - A partir de sa déclaration.
  - Jusqu'à la fin du bloc l'ayant déclarée.
  - Egalement dans les sous blocs.
- Illustration

```
int i;
scanf("%d", &i);
if (i < 0)
{
    int j; // déclaration de j ici
    j = i;
    i = 0; // i est visible dans ce sous bloc
    printf("i doit être au minimum égal à 0.\n");
} // j n'est plus visible au delà
// erreur de compilation : j n'est pas visible
if (j < 0)
    printf("L'ancienne valeur saisie %d est perdue.\n", j);
```

Variables locales

Visibilité

- Variables de même type et de même nom dans des blocs séparés
  - Ce sont des variables distinctes !
  - Emplacements mémoires différents.
- Exemple

```
void afficher_table_multiplication(long valeur)
{
    long i; // variable i de afficher_table_multiplication
    printf("Table de multiplication de %ld\n", valeur);
    for (i = 1; i <= 10; i++)
        printf("%ld * %ld = %ld\n", i, valeur, i * valeur);
    printf("\n");
}

int main(void)
{
    long i; // variable i de main, distincte de la précédente !
    for (i = 1; i <= 10; i++)
        afficher_table_multiplication(i);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Variables locales

Visibilité avec les blocs imbriqués - masquage

- Il y a masquage lorsque
  - un bloc et un sous bloc déclarent une variable de même nom.
  - La deuxième variable masque la première dans le sous bloc.
- Illustration

```
long i; // première variable i
i = 5;
{
    long i; // nouvelle variable i
    i = 10;
    printf("Deuxième variable:%ld\n", i); // affiche 10
}
printf("Première variable:%ld\n", i); // affiche 5
```

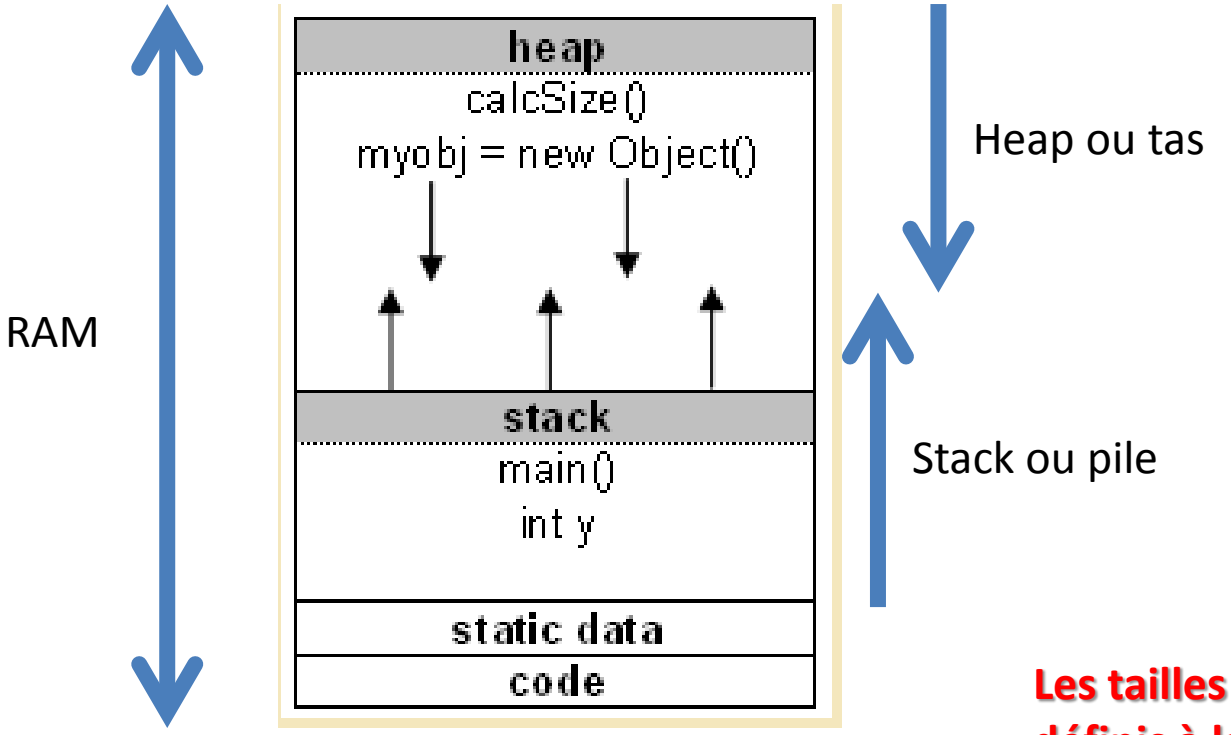
Variables locales

Durée d'existence

- A quel moment une variable commence-t-elle à exister ?
  - Une variable correspond à un emplacement en mémoire.
  - Elle commence donc à exister quand son emplacement est défini.
  - Les variables sont créées pendant l'exécution du programme.

- Pour les variables locales (et les paramètres)
  - De la mémoire est attribuée aux variables au début de l'exécution de la fonction ou du bloc de code.
  - Cette mémoire est libérée à la fin de la fonction ou du bloc de code
    - Elle redevient ainsi disponible pour d'autres variables.
  - Tout cela est géré automatiquement par le programme compilé.

Segments de la mémoire RAM



Les tailles du stack et du heap sont définis à la compilation

Variables locales

Durée d'existence - illustration

```
void afficher_table_multiplication(long valeur)
// « valeur » créée lors de l'appel de la fonction
{
    // « i » créée lors de l'exécution de cette ligne :
    long i;
    printf("Table de multiplication de %ld\n", valeur);
    for (i = 1; i <= 10; i++)
    {
        // « produit » créée lors de l'exécution de cette ligne :
        int produit;
        produit = i * valeur;
        printf("%ld * %ld = %ld\n", i, valeur, produit);
    } // « produit » disparaît sur cette ligne
    printf("\n");
} // « i » disparaît sur cette ligne
```

La variable « produit » sera créée et détruite à chaque itération, donc 10 fois au cours de l'exécution de la fonction

Enchaînement des appels de fonction

Une fonction peut en appeler une autre

- Exemple

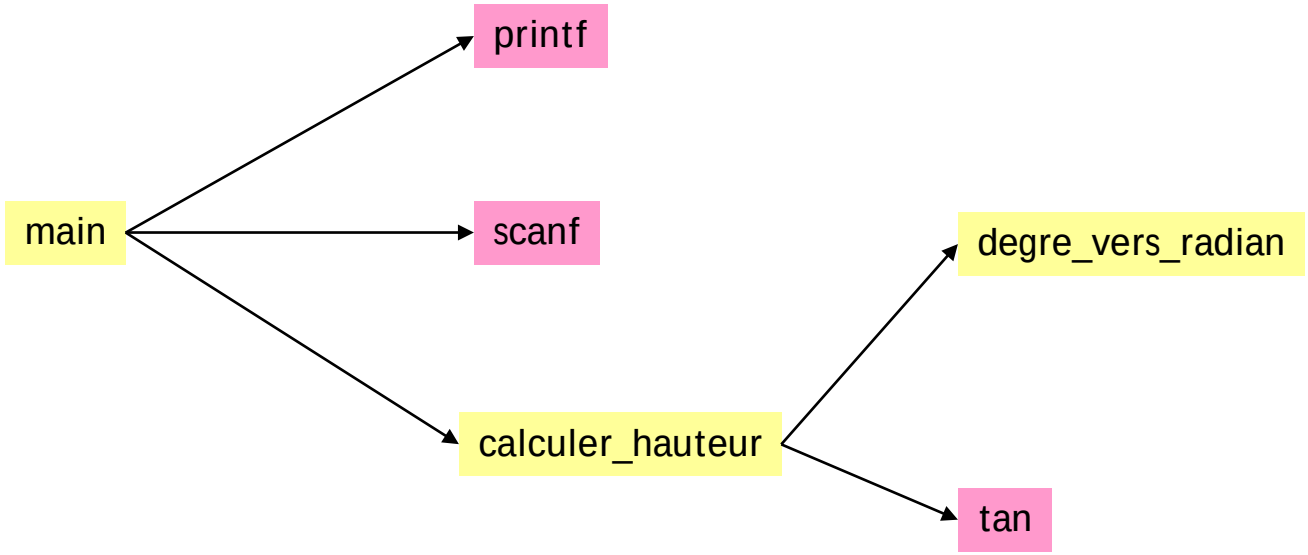
```
double degre_vers_radian(double angle_degre)
{
    return angle_degre * M_PI / 180.0;
}

double calculer_hauteur(double distance, double
  angle_degre)
{
    double angle_radian;
    angle_radian = degre_vers_radian(angle_degre);
    return distance * tan(angle_radian);
}
```

Enchaînement des appels de fonction

Décomposition hiérarchique d'un programme

- Les **fonctions permettent de découper un programme**
  - Jusqu'à un niveau de fonctions élémentaires.
  - Diviser un programme complexe en petites fonctions simples
  - Outil très puissant pour simplifier la programmation.



heig-vd

Appel de fonctions

Organisation du programme en mémoire et exécution

```
Adr. Code
1000 int main(void)
     {
1002   printf("Valeur : %d\n", 123);
1006   printf("Fin du programme\n");
100A   system("PAUSE");
100D   return EXIT_SUCCESS;
     }

       // autres fonctions
       . . .

1020 int printf(const char * format, ...)
     {
       int i, j;
1021   // implémentation interne
     }
```

1. Lorsque le programme démarre, l'adresse courante d'exécution vaut 1000 (exemple). Cette adresse se trouve dans le registre "Program Counter).

2. Avant d'appeler printf, il faut placer les paramètres dans la mémoire (pile/stack) pour permettre ensuite à printf de les utiliser.

3. Lorsque main appelle printf, l'adresse d'exécution courante prend la valeur 1020.

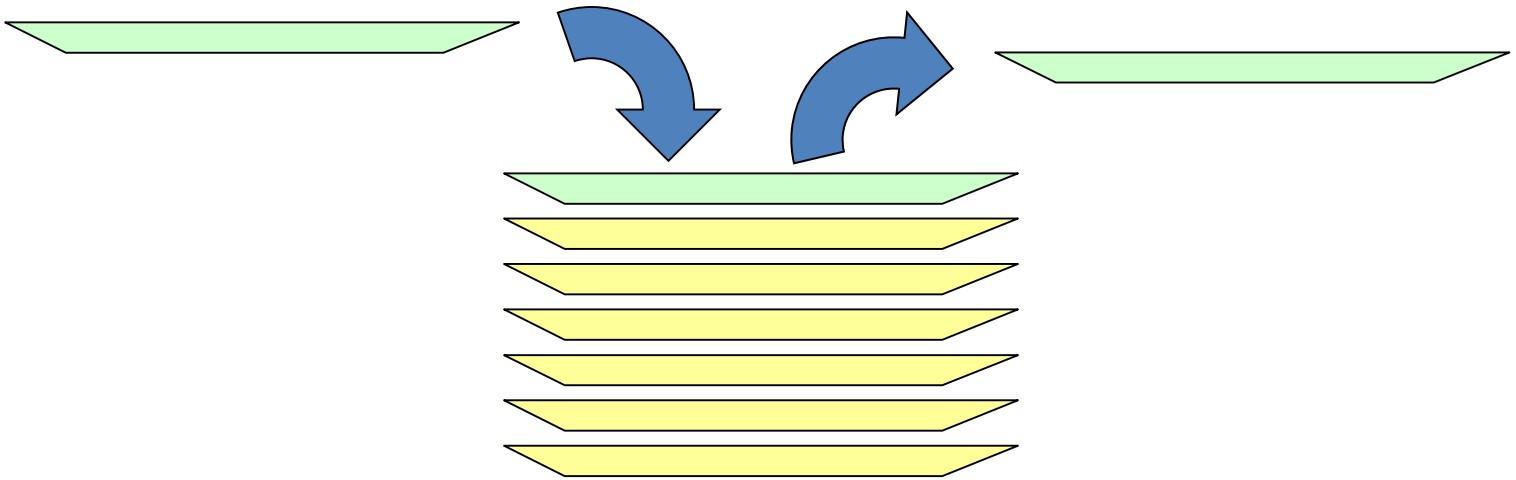
4. printf s'exécute et utilise les paramètres.

5. Lorsque printf est terminée, l'adresse d'exécution courante doit désigner la prochaine instruction du programme.

Appel de fonctions

La pile - principe

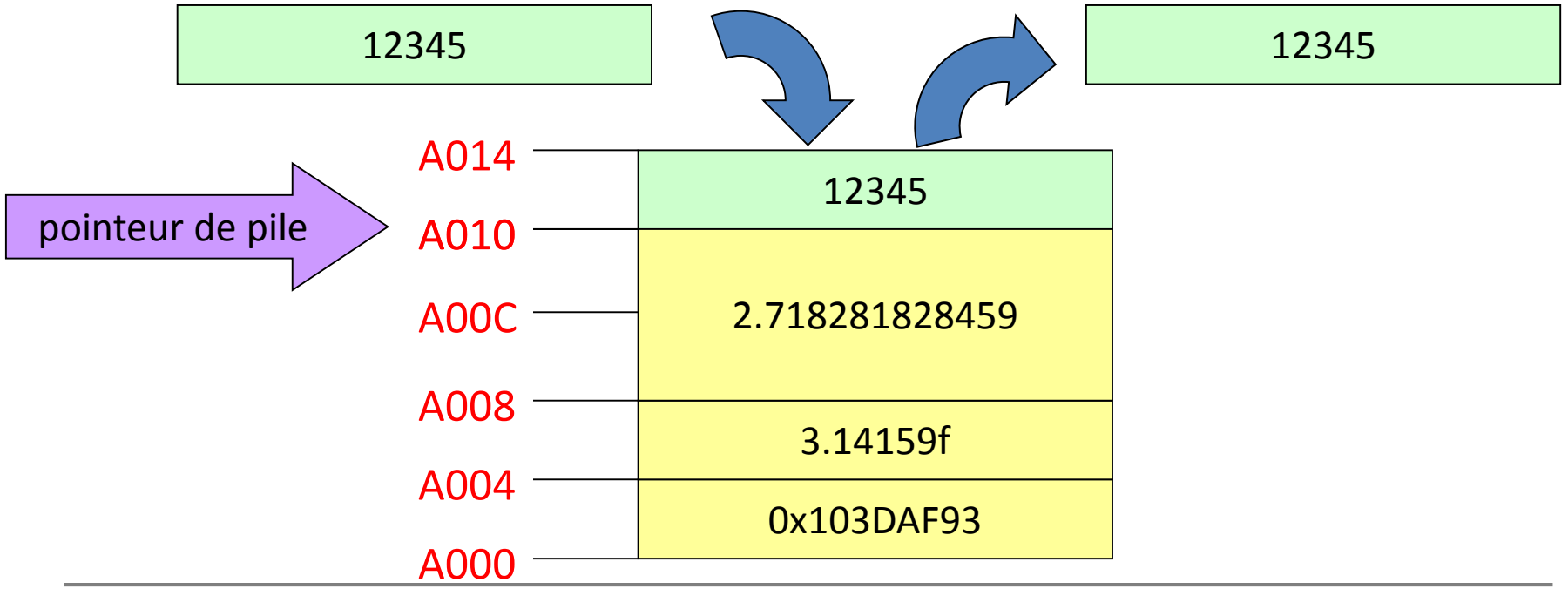
- Similarité avec une pile d'assiettes :
  - Les premières assiettes reprises sont les dernières posées



Appel de fonctions

La pile - principe

- Utilisation dans la pratique
  - Des valeurs sont placées sur la pile.
  - Les premières valeurs récupérées sont les dernières qui ont été placées (Fisrt In, Last Out => FILO).



Appel de fonctions

La pile – mise en œuvre dans l'ordinateur

- La plupart des CPU ont un registre dédié (à la pile)
  - Il contient l'adresse courante de la pile.
  - Il est souvent appelé SP (Stack Pointer)
- La plupart des CPU ont des instructions assembleur dédiées
  - PUSH (pousser)
    - Ecrit une valeur sur la pile.
    - Incrémente automatiquement le pointeur de pile.
    - De façon très efficace.
  - POP (prélever)
    - Décrémente automatiquement le pointeur de pile.
    - Lit la valeur sur la pile.
    - De façon très efficace.

Quelques registres d'un microprocesseur

- Les accumulateurs
  - Ce sont des mémoires rapides, à l'intérieur du microprocesseur, qui permettent à l'UAL de manipuler des données à vitesse élevée.
- Le compteur ordinal (pointeur de programme (Program Counter PC))
  - Pointe sur l'adresse où se trouve le programme
- Le registre de status
  - Chaque bit indique un état du processeur (ex.: interruption, retenue, etc)
- Le pointeur de pile (stack pointer SP)
  - Pointe sur l'adresse où se trouve la pile

Appel de fonctions

Utilisation de la pile lors de l'appel d'une fonction

- **Avant l'appel d'une fonction**
  - Les paramètres effectifs sont évalués.
  - Leurs valeurs sont empilées sur la pile.
- **Exécution de l'appel de fonction**
  - L'adresse de retour est empilée.
  - L'adresse d'exécution courante est changée vers la fonction à exécuter.
- **A l'entrée de la fonction**
  - Le pointeur de pile est déplacé pour réserver de la place pour les variables locales.

Appel de fonctions

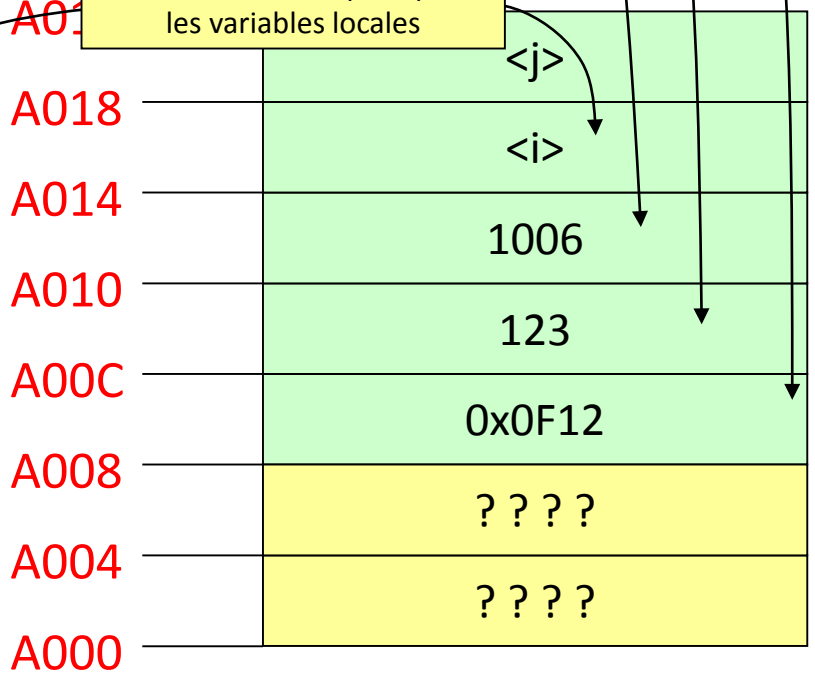
Exécution d'un appel

```
Adr. Code
1000 int main(void)
     {
1002   printf("Valeur : %d\n", 123);
1006   printf("Fin du programme\n");
100A   system("PAUSE");
100D   return EXIT_SUCCESS;
     }

       // autres fonctions
       . . .

1020 int printf(const char * format, ...)
     {
       int i, j;
1021   // implémentation interne
     }
```

1. Empilement du premier paramètre (pointeur).
2. Empilement du deuxième paramètre.
3. empilement de l'adresse de retour
4. Branchement au début de la fonction
5. réservation d'espace pour les variables locales



Appel de fonctions

Utilisation de la pile lors du retour d'une fonction

- **Exécution du retour de fonction**
  - Le pointeur de pile est déplacé pour libérer l'espace des variables locales.
  - L'adresse de retour est dépilée.
  - L'adresse d'exécution courante est changée vers l'adresse de retour qui a été dépilée.
- **Au retour dans la fonction appelante**
  - Le pointeur de pile est déplacé pour libérer l'espace des paramètres.

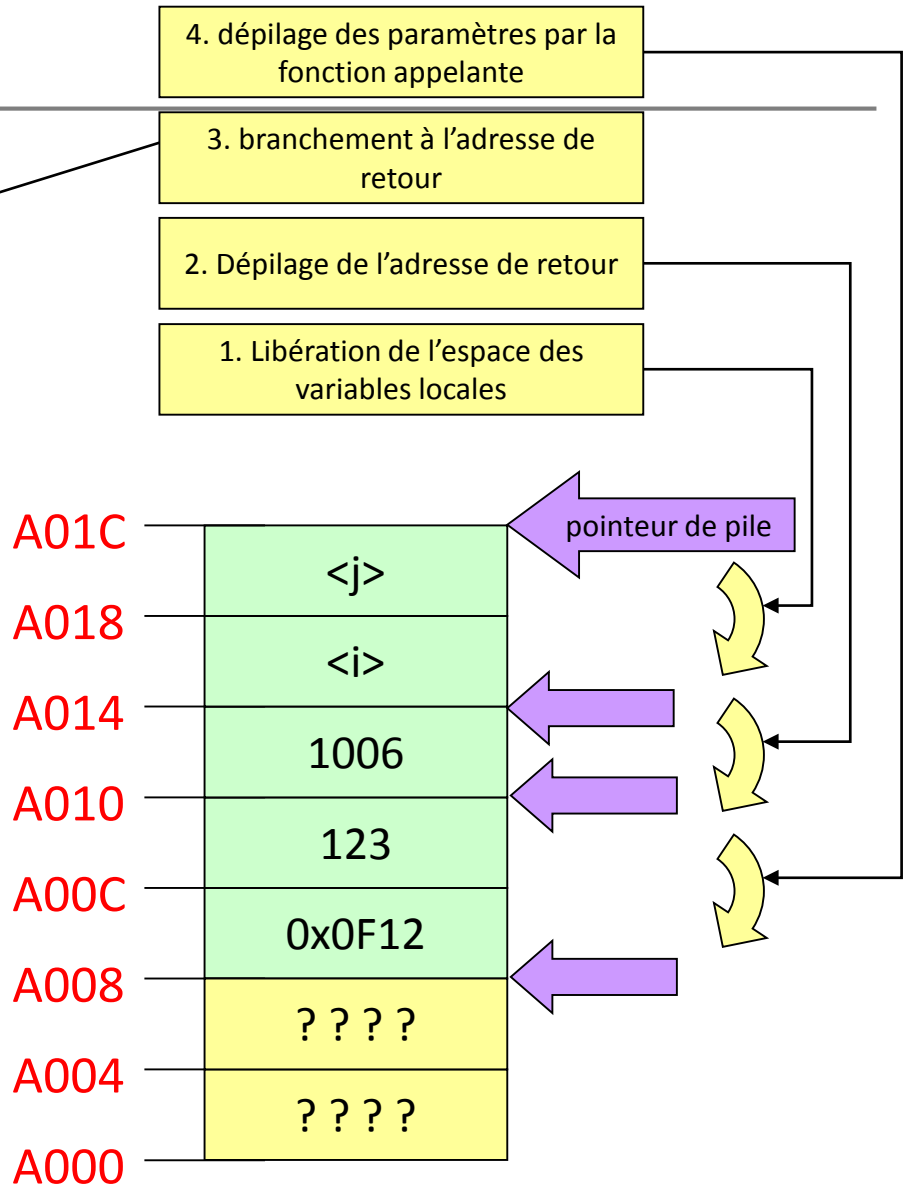
Appel de fonctions

Exécution d'un retour de fonction

```
Adr. Code
1000 int main(void)
     {
1002   printf("Valeur : %d\n", 123);
1006   printf("Fin du programme\n");
100A   system("PAUSE");
100D   return EXIT_SUCCESS;
     }

       // autres fonctions
       . . .

1020 int printf(const char * format, …)
     {
       int i, j;
1021   // implémentation interne
     }
```



Appel de fonctions
Conclusion

- La mémoire de travail
  - Elle est gérée selon le principe de la pile.
  - C'est un principe simple et efficace.
- Ce principe permet facilement de
  - Passer la valeur de paramètres à une fonction.
  - Trouver des emplacements mémoire libres pour les variables locales.
  - Savoir à quel adresse l'exécution continue après l'exécution de la fonction.
- Les compilateurs du langage C
  - Génèrent automatiquement le code nécessaire.
  - Nous permettent d'écrire nos fonctions sans trop s'en préoccuper.
  - La compréhension est cependant utile pour analyser les problèmes.

Fonctions avec un nombre variable de paramètres

Exemples

- Nous avons souvent utilisé les fonctions
  - printf
  - scanf
- Ces fonctions sont particulières
  - Elles acceptent un nombre variable de paramètres!

```
// 1 paramètre
printf("Hello world !\n");
// 2 paramètres
printf("Table de multiplication de %ld\n",
  valeur);
// 3 paramètres
printf("%ld * %ld = %ld\n", i, valeur, produit);
```

  - Pouvons nous aussi créer de telles fonctions ?

Fonctions avec un nombre variable de paramètres
Comment les déclarer

- Exemple

```
int printf(const char * format, ...);
double moyenne(int nombre_valeurs,...);
```

- Règles
  - Une fonction avec nombre variable de paramètres doit comporter au moins un paramètre fixe.
    - Ce paramètre fixe étant utilisé pour trouver les autres…
  - La liste variable de paramètres doit toujours figurer en dernière position parmi les paramètres formels.
  - La liste variable est représenté par 3 points (…)
- Exemple d'utilisation

```
printf("max:%lf\n", moyenne(3, 1., 2., 3.));
```

heig-vd

Fonctions avec un nombre variable de paramètres

Comment récupérer leur valeur dans la fonction

- Le langage C est fourni avec une bibliothèque spéciale
  - `#include <stdarg.h>`
- Elle définit un type
  - A utiliser pour traiter les listes variables de paramètres.
  - Nom : **`va_list` Pointeur sur la liste des arguments**
- Et 3 macros pour récupérer la valeur des paramètres
- **`// 1ère macro`**
  - **`// prépare la variable de type va_list`**
  - **`va_start(va_list, parameter_1);`**
    - ***`va_list`*: Pointeur sur la liste des arguments**
    - ***`parameter_1`*: Nom du paramètre qui précède le 1er argument optionnel (qui définit le nombre de paramètres qui suivent)**

Suite

Fonctions avec un nombre variable de paramètres

Comment récupérer leur valeur dans la fonction

- .. Et les 2 autres macros pour récupérer la valeur des paramètres

- **// 2ème macro**

```
// permet de récupérer UN argument(paramètre)
// A appeler plusieurs fois, pour récupérer plusieurs arg
type parametre = va_arg(va_list, type);
```

  - ***va_list***: **Pointeur sur la liste des arguments**
  - ***type***: **type du paramètre optionnel (ex. int, double, etc.)**

- **// 3ème macro**

```
// doit être appelé à la fin
va_end(va_list)
```

  - ***va_list***: **Pointeur sur la liste des arguments**

Fonctions avec un nombre variable de paramètres

Exemple de fonction avec nombre de paramètres variables (1/2)

```
double moyenne(int nombre_valeurs, ...) // Attention, les 3 points sont obligatoires
{
    if (nombre_valeurs > 0)
    {
        // déclaration de la variable de type va_list :
        va_list arglist;
        int i;
        double somme = 0.0;
        // préparer la variable arglist :
        va_start(arglist, nombre_valeurs);
        for (i = 0; i < nombre_valeurs; i++)
        {
            // récupérer un argument :
            double argument = va_arg(arglist, double);
            somme += argument;
        }
        // important pour garantir un bon fonctionnement :
        va_end(arglist);
        return somme / nombre_valeurs;
    }
    else
        return 0.0;
}
```

Fonctions avec un nombre variable de paramètres

Exemple de fonction avec nombre de paramètres variables (2/2)

```
//Prototype de fonction à nombre de paramètres variables
double moyenne(int nombre_valeurs, ...);

int main(void)
{
   double result;

   result = moyenne(3, 15.0, 25.0, 50.0);  // avec 4 paramètres
   printf("\nLa moyenne est de %lf\n", result);

   result = moyenne(4, 15.0, 25.0, 50.0, 100.0); // avec 5 paramètres
   printf("\nLa moyenne est de %lf\n", result);

   /* Pour terminer, non standard!!! */
   printf ("\n\nFin du programme...");
   system ( "pause" );
   return EXIT_SUCCESS;
}
```

Fonctions avec un nombre variable de paramètres

Comment savoir si tous les paramètres ont été utilisés ?

- stdarg ne fournit aucun moyen pour détecter la fin de la liste de paramètres.
- En conséquence, le programmeur de la fonction doit créer son propre mécanisme
  - Dans l'exemple précédent :
    - Le nombre de valeurs passées en argument est indiqué dans le premier paramètre de la fonction.
  - printf, scanf
    - C'est la chaîne de format qui indique le nombre et le type des paramètres.

Fonctions avec un nombre variable de paramètres

Règles de passage de paramètres

- Pour les paramètres de type entier
  - char, short sont convertis en int (sauvé sur 4 bytes)
  - long reste au format long.
- Pour les paramètres de type réel
  - Ils sont convertis en double.
- Fonctionnement interne
  - va_arg se promène sur la pile.
  - Attention à récupérer les paramètres avec le bon type.
    - Sinon, va_arg se décale par rapport au contenu de la pile.
- En résumé: utilisation dangereuse

Les fonctions inline

Pour améliorer les performances

- Une fonction peut être déclarée **inline**
  - Signifie littéralement « en ligne »

```
inline int max( int a , int b )
{
    if( a > b )
        return a;
    else
        return b;
}
```

- Effet de **inline**
  - Le compilateur remplace les appels à la fonction par son code complet.
  - Gain : moins de gestion au niveau de la pile, donc plus rapide.
  - Coût : Pour chaque appel de fonction, le code complet est re-généré.
    - Donc, coût au niveau de la taille du code.
- Intéressant lorsque les performances sont critiques.


Exercices
Créer des fonctions

Les fonctions
Compléments

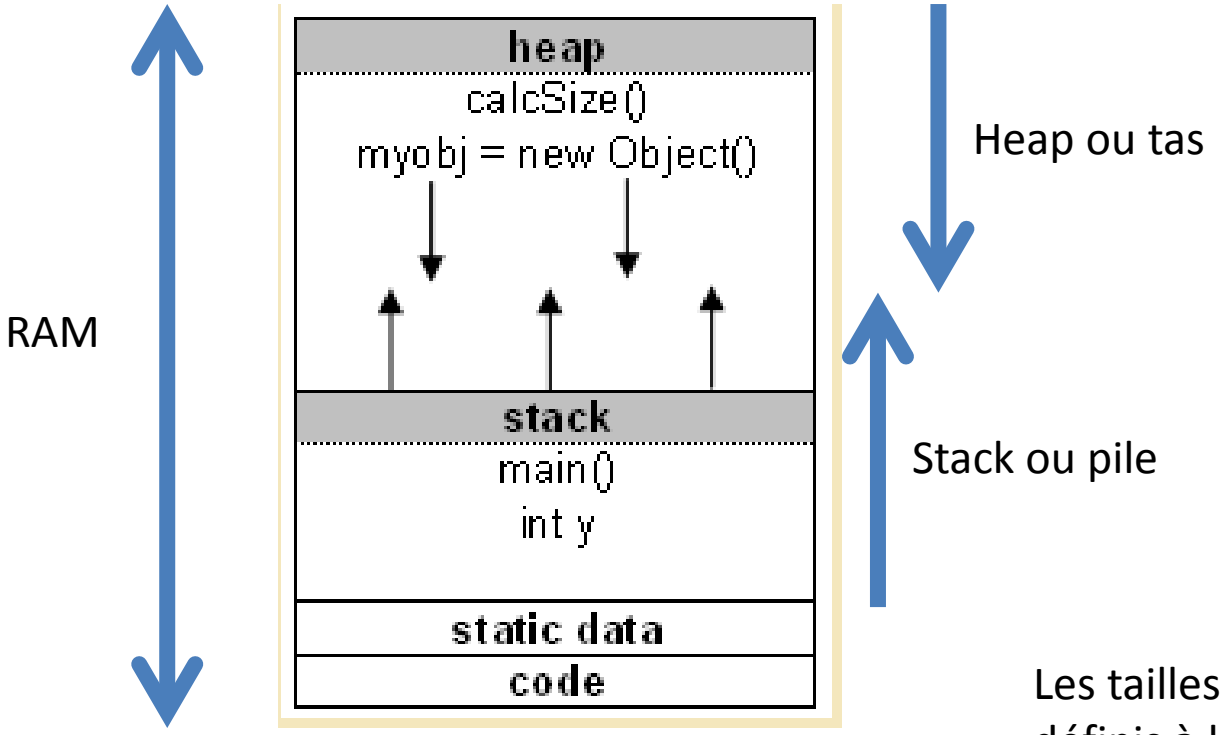
- Comment mémoriser des états globaux à un programme ?
  - Les variables globales
- Comment passer un paramètre par variable (appelé aussi par référence) en C ?
  - Le passage de paramètres par adresse

Les variables globales

Principe

- Nous avons vu jusqu'à présent les variables locales
  - Visibles seulement à l'intérieur d'une fonction ou d'un bloc.
  - Créées au début du bloc, détruites à la fin
- Le langage C permet de créer des variables globales
  - Visibles par plusieurs fonctions.
  - Existent aussi longtemps que le programme est en mémoire.
  - Conservent leur valeur entre 2 appels consécutifs d'une fonction.
- Emplacement mémoire
  - Elles sont créées à un emplacement fixe en mémoire.
    - Appelé le segment statique.
  - Elles ne sont donc pas créées sur la pile.

Segments de la mémoire RAM



Les tailles du stack et du heap sont définis à la compilation

Les variables globales
Déclaration

- Une variable globale se déclare
  - Comme une variable locale.
  - La déclaration se trouve en dehors de toute fonction.
- Une variable globale est visible par toutes les fonctions se trouvant après sa déclaration.
- Initialisation automatique
  - Par défaut les variables globales sont initialisées à 0.
  - Par soucis de clarté, il est cependant préférable de toujours les initialiser explicitement.

Les variables globales

Exemple

```
double coefficient_global;

double f(double x)
{
    double resultat;
    resultat = coefficient_global * x;
    coefficient_global = x;
    return resultat;
}

int main(void)
{
    double resultat;
    coefficient_global = 1.0;
    resultat = f(2) + f(3);
    printf("%f\n", resultat); // affiche ???
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Résultat = 8
(2 *1) + (2 * 3)

Les variables globales
Utilité

- Mémorisation d'un état au-delà de la durée d'une fonction
- Exemple : détecter un flanc sur une entrée tout ou rien.

résultat : 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

```
int etat_precedent = 0;

int flanc_montant_detecte()
{
    int flanc;
    flanc = (entree == 1 && etat_precedent == 0);
    etat_precedent = entree;
    return flanc;
}
```

Les variables globales

Un premier **danger des variables globales** – les effets de bord

- Un effet de bord survient lorsque le résultat d'une fonction
  - ne dépend pas seulement de ses paramètres d'entrée.
  - mais également de variables d'état (globales) du programme.
- Effets de la modification d'une variable d'état
  - Avec un même jeu de paramètres, le résultat de la fonction est différent.
  - La fonction ne correspond plus à la notion mathématique de fonction.
- Dans l'exemple précédent
  - La modification de coefficient_global change les valeurs retournées par la fonction f.

Les variables globales

Un premier **danger des variables globales** – les effets de bord

- Conséquences
  - Peut rendre un programme particulièrement difficile à analyser.
- Illustration : la perte de commutativité de l'addition
  - Analyse sur l'exemple précédent.
  - f(2) + f(3) = 2 * 1 + 3 * 2 = 8
  - f(3) + f(2) = 3 * 1 + 2 * 3 = 9
  - Donc, f(2) + f(3) != f(3) + f(2).
- Influence sur la lisibilité
  - La lecture des programmes devient très délicate.
  - Nos réflexes élémentaires sont remis en question.

Les variables globales
Initialisation des variables globales

- Pour initialiser une variable globale
  - La syntaxe habituelle de l'initialisation des variables est valable.
- Exemple

```
double coefficient_global = 1.0;
```

- L'initialisation est effectuée avant l'entrée dans « main ».

Les variables globales

Utilisation pour passer des paramètres et des résultats

- Utiliser des variables globales pour échanger de l'information ?
- On pourrait remplacer les paramètres et les résultats des fonctions par des variables globales.

```
double resistance, courant, tension;

void calculer_loi_ohm_tension(void)
{
    tension = resistance * courant;
}

int main(void)
{
    resistance = 220;
    courant = 0.150;
    calculer_loi_ohm_tension();
    printf("Tension : %f\n", tension);
    return EXIT_SUCCESS;
}
```

Les variables globales

Utilisation pour passer des paramètres et des résultats - analyse

- Lisibilité du code
  - Les paramètres d'entrée et de sortie de la fonction **`calculer_loi_ohm_tension`** n'apparaissent plus explicitement.
  - Il faut lire le code pour comprendre l'interface de la fonction.
  - Il y a donc des dépendances cachées.
    - **Programmation spaghetti.**
  - Si on oublie de préparer un paramètre, le compilateur ne dit rien !
  - Même sur un exemple simple, c'est déjà très mauvais.
- Occupation mémoire
  - Les paramètres de fonction occupent de la mémoire temporairement pendant l'exécution de la fonction.
  - Les variables globales, elles, sont présentes en permanence.
  - Donc, passer les paramètres par des variables globales coûte cher en espace mémoire.

Les variables globales

Utilisation pour passer des paramètres et des résultats - conclusion

1. **Ne pas utiliser les variables globales pour échanger de l'information entre fonctions.**

2. Réserver l'usage des variables globales à la mémorisation d'informations d'état qui sont conceptuellement partagées.

3. **Ne pas en abuser.**

Passage de paramètre par adresse

Rappel – le passage par valeur

```
void incrementer(int entier)
{
    // entier: variable locale contenant une copie de i !
    entier = entier + 1;
}

int main(void)
{
    int i;
    i = 0;
    incrementer(i);
    // i vaut toujours 0 !
    printf("Nouvelle valeur de i: %d\n", i);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Passage de paramètre par adresse

Une imitation du passage par variable (par référence)

- Comment modifier le contenu d'une variable passée en paramètre ?
- Nous avons déjà vu un exemple
  - La fonction scanf
- Exemple

```
int i;
scanf("%d", &i);
```

- Fonctionnement
  - L'expression &i signifie adresse de i.
  - On ne passe donc pas la valeur de i, mais son adresse à scanf.
  - Connaissant l'adresse de la variable, scanf peut y accéder
  - Et donc la modifier !

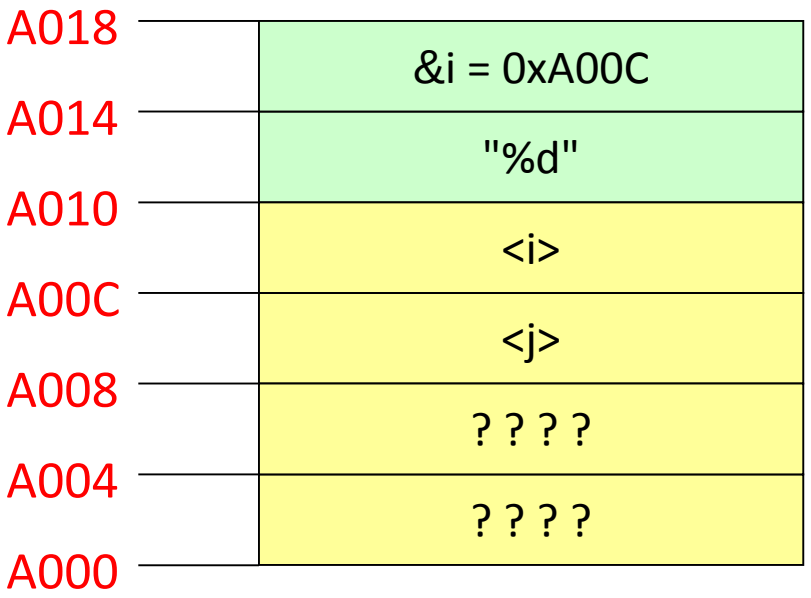
Passage de paramètre par adresse

Une imitation du passage par variable - illustration

- Programme exemple

```
int j, i;
scanf("%d", &i);
```

- Que se passe-t-il à l'appel de scanf ?
  - Empilement de "%d" (en réalité, pointeur de tableau de char)
  - Empilement de l'adresse de i
- Connaissant l'adresse de i
  - scanf peut en modifier le contenu !



Passage de paramètre par adresse

Une imitation du passage par variable

- Comment déclarer un paramètre passé par adresse ?
  - Utiliser la syntaxe des pointeurs
  - Il suffit de rajouter une * après le nom du type.

```
void incrementer(int * p_entier)
```

  - On peut rajouter un préfixe au nom de la variable
    - Pour bien montrer que ce n'est pas un entier, mais un pointeur !
  - Dans la pratique, une adresse ou un pointeur, c'est un entier !
- Comment manipuler la variable dont on a l'adresse
  - Il existe en C l'opérateur *****.
  - Il **déréférence** le pointeur (ou l'adresse).

```
*p_entier // désigne la variable à cette adresse
```

  - **`*p_entier`** est la valeur situé à l'adresse du pointeur **`p_entier`**.

Passage de paramètre par adresse

Une imitation du passage par variable - un exemple qui fonctionne

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

int main(void)
{
    int i;
    i = 0;
    incrementer(&i);
    // i vaut maintenant 1
    printf("Nouvelle valeur de i: %d\n", i);
    system("PAUSE");
   return EXIT_SUCCESS;
}
```

Passage de paramètre par adresse

Une imitation du passage par variable – les pièges

- Appel de la fonction
  - Attention à bien passer une adresse et non pas un entier !

```
int i;
i = 0;
incrementer(i); // compile, mais corruption mémoire !
```

- Exécution de la fonction
  - Attention à bien déréférencer le pointeur.

```
p_entier = p_entier + 1;
// incrémente l'adresse. ex : 0xA00C -> 0xA00D
// mais sans aucun effet sur la variable pointée
```

heig-vd

Qu'affiche ce programme ?

```
int test(int * a, int b, int * c, int * d)
{
    b = *a;
    *a = *a + 3;
    *c = b + 7;
    c = d;
    return *c;
}

int main(void)
{
    int p = 0, q = 10, r = 20, s = 30, t = 40;
    t = test(&p, q, &r, &s);
    printf("p:%d, q:%d, r:%d, s:%d, t:%d.\n", p, q, r, s, t);
    _getch();
   return EXIT_SUCCESS;
}
```

heig-vd

Qu'avons-nous appris ?

- Comment créer et utiliser une fonction
  - Prototype et définition
  - Avec et sans paramètres.
  - Avec et sans résultat.
- Les paramètres
  - Par valeur.
- Les variables
  - Locales
  - Globales, et les risques associés.
- Le fonctionnement interne
  - Utilisation de la pile lors des appels de fonction

heig-vd

Vos questions


Exercices
Fonctions et
variables