Plan

1 Fonctions

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

A quoi sert une fonction

Les fonctions servent à partitionner les gros traitements en traitements plus petits.

Elle permettent de définir des briques de calcul qui sont :

- identifiables facilement,
- conçues pour réaliser un traitement précis,
- réutilisables.

Remarque

Les programmes sont plus lisibles et donc plus faciles à maintenir.

Fonction : une brique de calcul paramétrée

On veut écrire un programme qui affiche la décomposition en facteurs premiers d'un nombre entier.

```
1  int main() {
2      int x,y,i,k;
3      printf("donnez un entier : ");
4      scanf("%d",&x);
5      y = x; i = 2;
6      while (x>=i) {
7          //calcul de iprem vrai si i premier, faux sinon
8          if (iprem && x%i==0) {
9              //calcul de k plus grande puissance de i divisant x
10             printf("%d puissance %d divise %d\n", i, k, y);
11             //calcul de d valeur de i puissance k
12             x = x/d;
13         }
14         i++;
15     }
16     return 0;
17 }
```

Fonction : une brique de calcul paramétrée

Deux solutions :

✗ on complète le code directement par les instructions de calcul correspondantes.

✓ on complète le code par l'appel à trois briques de calcul paramétrées.

```
1  _Bool premier(int n) {
2      _Bool prem;
3      if (n==0 || n==1) prem = 0;
4      else if (n==2) prem = 1 ;
5      else {
6          int i = 2;
7          while(i*i<=n && n%i!=0) i++;
8          if(n%i==0) prem = 0;
9          else prem = 1;
10     }
11     return prem;
12 }
```

Fonction : une brique de calcul paramétrée

```
1 int plusGrandePuissanceDiv(int i, int x) {
2   int p = i, j = 1;
3   while(p<=x && x%p==0) {
4     j++;
5     p = p*i;
6   }
7   return j-1;
8 }
```

```
1 int puissance(int x, int n) {
2   int p = 1;
3   for (int j=1;j<=n;j++)
4     p = p*x;
5   return p;
6 }
```

Fonction : une brique de calcul paramétrée

Le programme s'écrit donc :

```
1  int main()
2  {
3      int x,y,i,k;
4      printf("donnez un entier : ");
5      scanf("%d", &x);
6      y = x; i = 2;
7      while (x>=i) {
8          if (premier(i) && x%i==0)
9          {
10             k = plusGrandePuissanceDiv(i,x);
11             printf("%d puissance %d divise %d\n", i, k, y);
12             x = x / puissance(i,k);
13         }
14         i++;
15     }
16     return 0;
17 }
```

Qu'est-ce qu'une fonction

Définition

Une fonction est une description formelle d'un calcul en fonction de ses arguments

Exemple

Si on vous donne deux variables entières a et b décrivez les instructions du langage nécessaires pour calculer a^b.
$\Rightarrow$ cela rejoint la notion d'algorithme

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Définition de fonctions C

Définition

```
typeRes nomFonction (type1 nom1, type2 nom2, ...)
{
  corps
}
```

- la signature :
  - `nomFonction` correspond au nom donné à la fonction (c'est un identificateur avec les mêmes règles de nommage que les variables)
  - `typeRes` est le type de données du résultat de la fonction
  - $\texttt{type}_i$ $\texttt{nom}_i$ sont les types de données et nom (c'est aussi un identificateur) de chaque paramètre formel
- le `corps` correspond aux instructions effectuées par la fonction en fonction des paramètres formels.

Noms de variables, fonctions et paramètres

- Un nom doit identifier un objet de manière unique.
- Les noms des fonctions, paramètres et variables ne peuvent être partagés au sein d'un même bloc.

Exemple

```
float f;              // déclaration de variable
int f() { ... }  // déclaration de fonction
```

provoquera l'erreur de compilation :
error : redefinition of 'f' as different kind of symbol

Exemple

```
int f(int a) { // déclaration de paramètre
   int a;      // déclaration de variable
}
```

provoquera l'erreur de compilation : error : redefinition of 'a'

Résultat d'une fonction C

Une fonction C renvoie un seul résultat : une valeur de `typeRes`

- cette valeur n'est pas nommée par un paramètre formel ;
- elle est définie dans le corps par l'instruction `return exp` qui calcule et renvoie la valeur de exp.

```
int f(...) {      // f doit renvoyer un entier
   ...
   return exp;    // la valeur de exp est renvoyee
}
```

- Le type de `exp` doit être `typeRes` (*sinon conversion implicite*)

```
double inverse(unsigned int x) {
  return 1/x ;   //renverra 0. pour tout x > 1
}
```

On peut définir une fonction qui ne renvoie rien (une procédure) en utilisant le type `void` : `void f(..) {...}`

Fonctions : paramètres formels

Les paramètres formels d'une fonction permettent :

- de définir le calcul à effectuer dans le corps de la fonction à partir de valeurs inconnues (mais typées) désignées par un nom et utilisables comme des variables ;
- des valeurs réelles seront transmises lors de l'appel et donneront une valeur initiale aux paramètres formels ;
- en dehors du bloc de définition de la fonction, ces paramètres n'existent pas (on ne peut pas accéder à leur valeur)

```
1 int f(int a, int b) {...}
2 _Bool g(double a) {...}
3 a = 12;                  // erreur a n'est pas declare
```

Une fonction peut n'avoir ni paramètres, ni résultat :

```
1 void bonjour() {
2   printf("Bonjour !\n");
3 }
```

Corps

Le corps d'une fonction est le bloc, après la signature, composé de :

- de la déclaration des variables locales (uniquement utilisables dans le corps de la fonction) ;
- de la séquence d'instructions exécutée lors de l'appel à la fonction dont une (ou plusieurs) instructions `return`

Définition

```
typeRes nomFonction (type1 nom1, type2 nom2, ...)
{
   // déclaration des variables
   // instructions
   // return exp ;
}
```

```
int plus(int x,int y){
  int z;
  z=x+y;
  return z;
}
```

L'instruction `return`

Définition

Dans une fonction, l'instruction `return` *exp* permet :

- d'arrêter l'exécution du corps
- de renvoyer la valeur de l'expression *exp* (après éventuelle conversion de type) au bloc appelant

Quand `return` apparaît plusieurs fois, le premier exécuté stoppe la fonction.

```
int impair (int x) {
  if (x%2 == 0) return 0;
  else return 1;
}
```

Attention, un `return` doit exister pour toute exécution du corps

```
int impair (int x){
  if (x%2 == 0) return 0;
}
```

↪ warning : non-void function does not return a value in all control paths

Définition n'est pas exécution

Attention

La définition d'une fonction est une description formelle d'un calcul, elle n'exécute rien toute seule : il faut *appeler* la fonction pour que les instructions du corps soient exécutées.

```
1 void bonjour() {
2     printf("Bonjour !\n");
3 }
```

Ce code ne fait que définir une fonction, il n'affichera rien.

↪ c'est l'instruction `bonjour();` qui exécutera ce code.

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Syntaxe d'un appel

Définition

`nomFonction(exp1, exp2,...);`

- `nomFonction` est le nom de la fonction appelée ;
- $\texttt{exp}_i$ sont des expressions : on les appelle les **paramètres effectifs**.

Attention

Les paramètres effectifs doivent correspondre en nombre et types à ceux de la définition de `nomFonction` (attention aux conversions implicites).

Quand la fonction renvoie un résultat, un appel peut être utilisé dans une expression complexe.

```
int plus(int x,int y) {
  int z;
  z=x+y;
  return z;
}
```

Exemples :

- `plus(2*3,7-2);`
- `printf("%d+%d=%d\n",3,5,plus(3,5));`
- `int val = 6*plus(3,5.);`
- `if(plus(a,b)>0) {...}`

Appel de fonction : sémantique

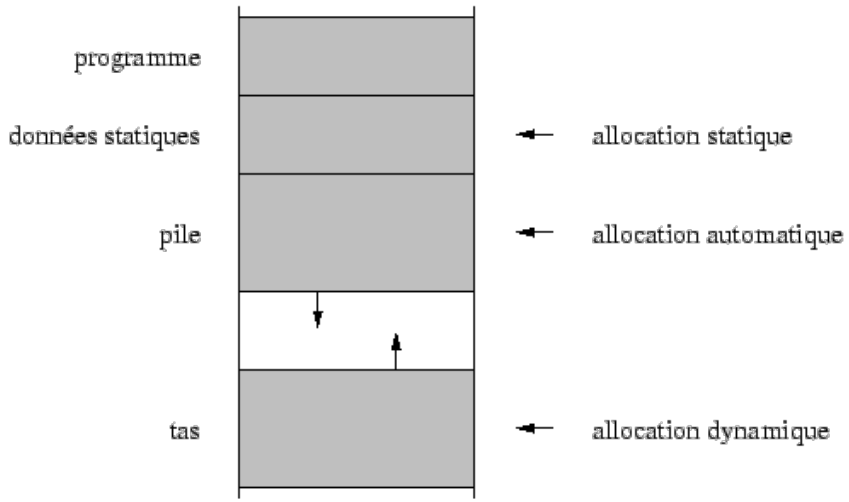
Evaluation d'un appel de fonction :

1. les valeurs v_i de chacune des $\texttt{exp}_i$ sont calculées
2. un environnement d'exécution de la fonction est préparé avec **création de variables** pour chaque paramètre formel initialisées avec les valeurs v_i calculées des paramètres effectifs
3. la fonction est exécutée
4. la valeur de retour de la fonction est mise à disposition (l'environnement est détruit)

Espace mémoire d'exécution

Lors de l'exécution d'un programme un espace mémoire est alloué au programme. On distingue différentes zones dans cet espace :

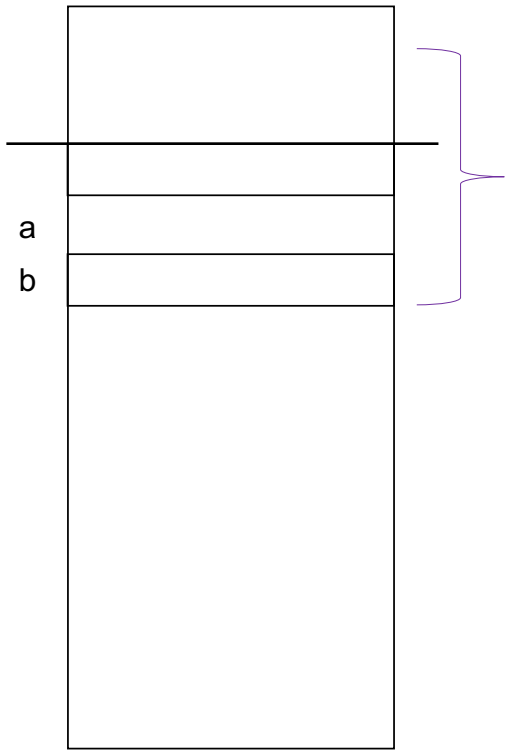
- le **segment de code** dédié aux instructions en langage machine du programme
- le **segment de données** dédiées aux données permanentes
- la **pile** dédiée aux données temporaires propres à chaque appel de fonction : **valeur de retour, paramètres formels, variables locales**.
- le **tas** qui permet la réservation durant l'exécution d'espace mémoire.



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

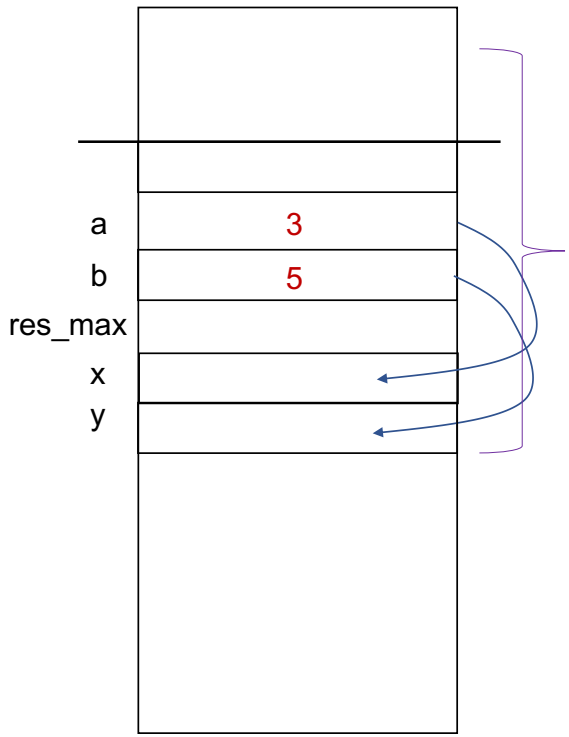
Pile à la ligne 10
environnement de la fct main



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

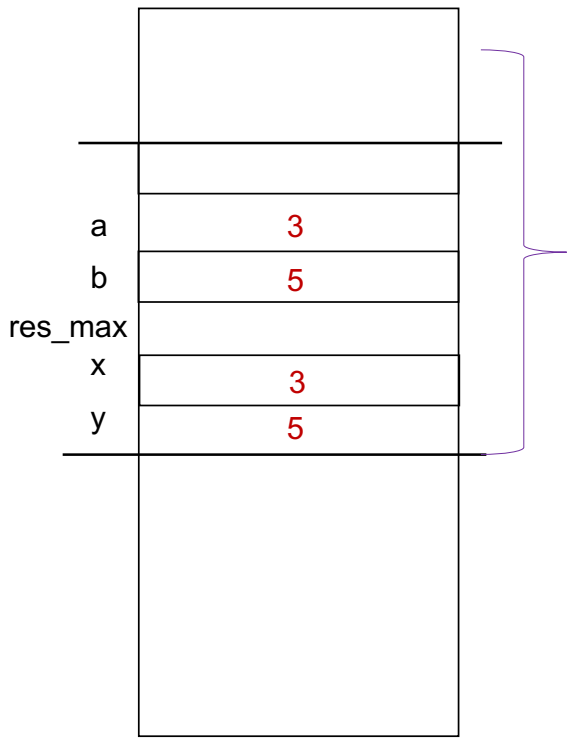
Pile à la ligne 13
préparation de l'appel à max



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

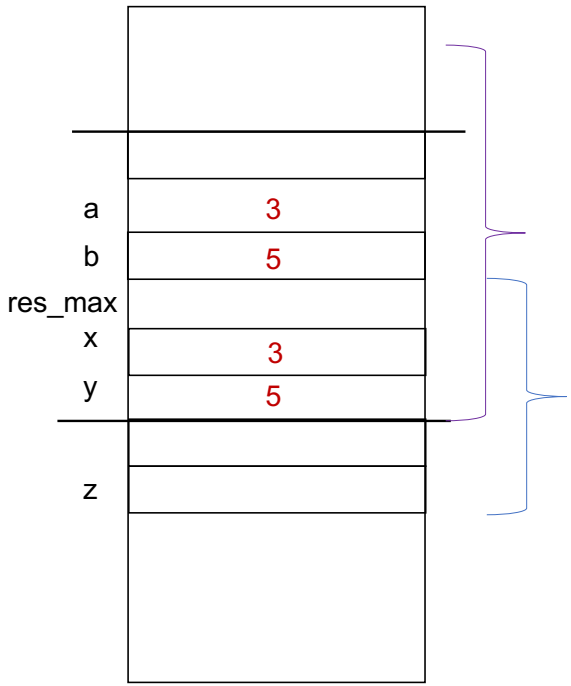
Pile à la ligne 13
appel à max



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

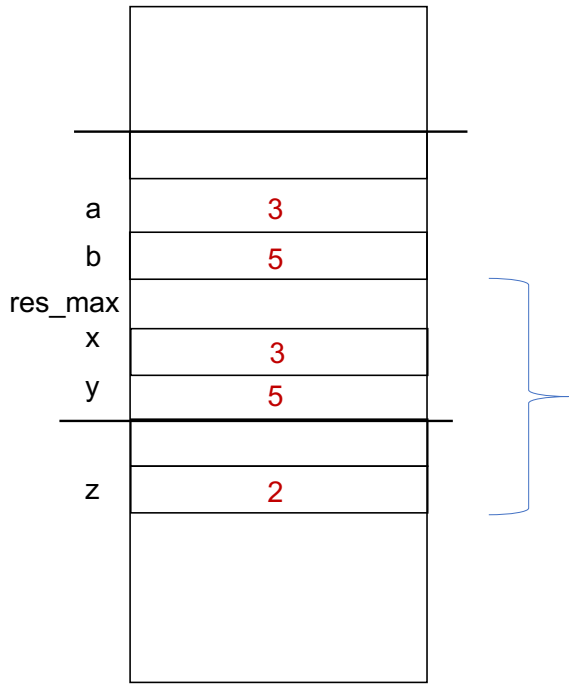
Pile à la ligne 3
création de l'env. de max



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

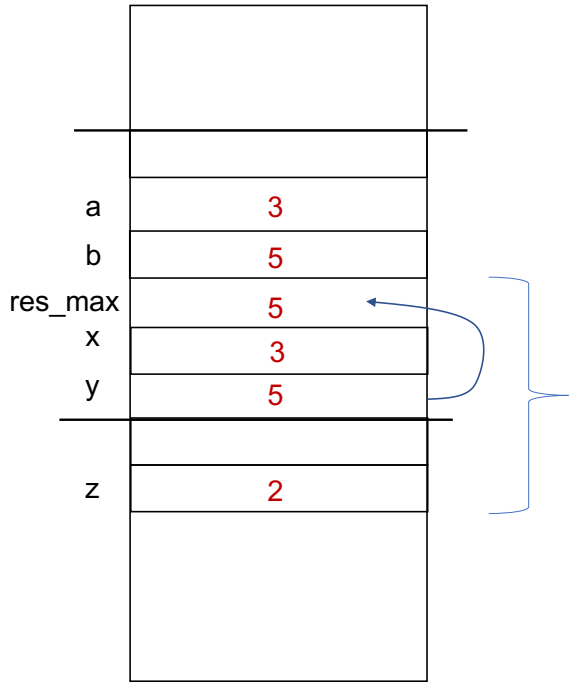
Pile à la ligne 4
exécution de max



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

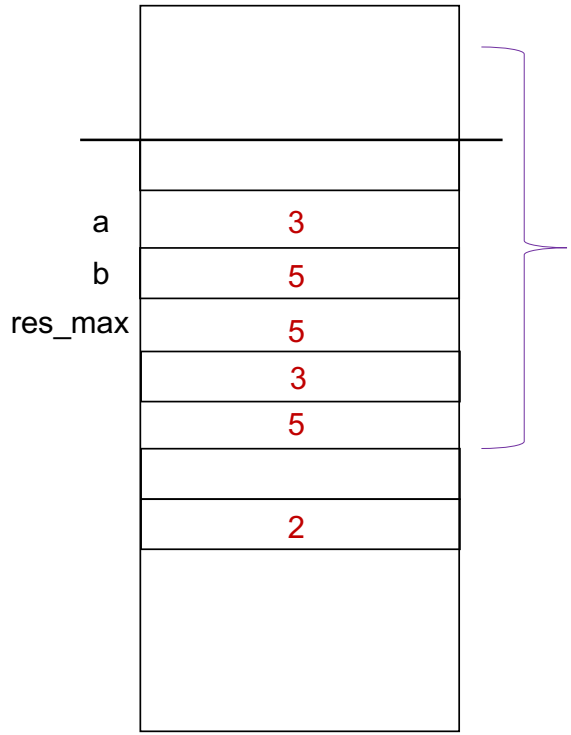
Pile à la ligne 6
copie du résultat



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

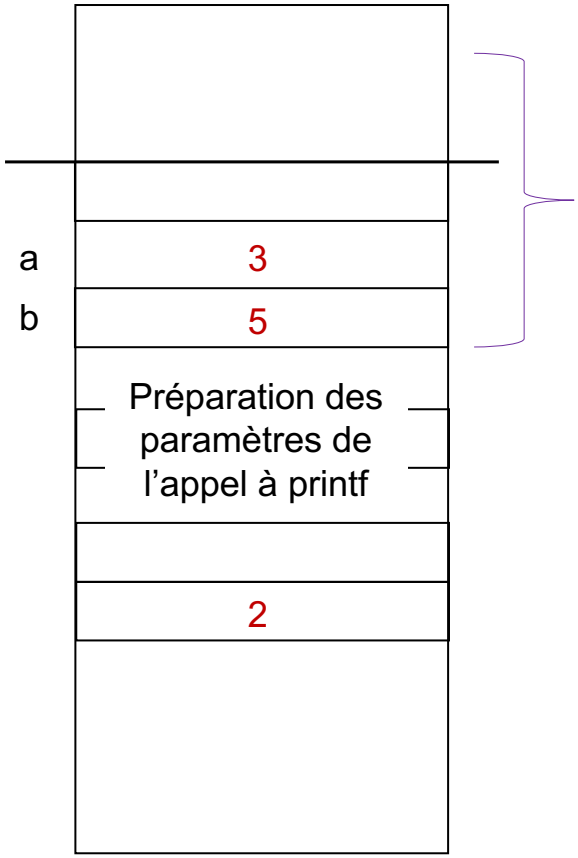
Pile à la ligne 13
retour à main



Exemple

```
1  #include <stdio.h>
2
3  int max(int x, int y) {
4     int z = 2 ;
5     if (x>y) return x;
6     else return y;
7  }
8
9  int main() {
10    int a, b;
11    a = 3;
12    b = 5;
13    printf("max=%d",max(a,b));
14    return 0;
15 }
```

Pile à la ligne 13
appel à printf



Passage de paramètres : danger !

Attention

La transmission des paramètres se fait par copie des valeurs des paramètres effectifs comme valeur initiale des paramètres formels.
⇒ Modifier la valeur des ces paramètres formels dans le corps de la fonction ne change en aucun cas les valeurs des paramètres effectifs.

```
1  void ech(int x, int y) {
2    int r;
3    r = x; x = y; y = r;
4  }
5
6  int main() {
7    int a, b;
8    a = 17; b = 13;
9    printf("a = %d b = %d \n", a, b);
10   ech(a,b);      // n'echange pas les valeurs de a et b
11   printf("a = %d b = %d \n", a, b);
12   return 0;
13 }
```

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Portée d'une variable

Définition

On appelle portée d'une variable, la portion du code où cette variable est connue et utilisable.

La portée dépend de la zone mémoire où la variable est créée.

Variables locales

Les variables locales sont par défaut (`auto`) créées dans la pile.

Leur portée est limitée au bloc (et dans les blocs imbriqués) dans lequel elles ont été déclarées et seulement à partir de la déclaration.

Exemple 1

Quelles sont les portées de a et de x ?

```
1 int main()
2 {
3     int a;
4     float x;
5     ...
6     ...
7     return 0;
8 }
```

Exemple 1

Quelles sont les portées de a et de x ?

```
1  int main()
2  {                    <-- debut du bloc
3      int a;           <-- debut de la portee de a
4      float x;         <-- debut de la portee de x
5      ...
6      ...              <-- utilisation possible de a et x
7      return 0;
8  }                    <-- fin de bloc et de portee de a et x
```

Code erroné

Est-ce que l'utilisation de a et de x est conforme à leur portée ?

```
1  int main()
2  {
3      int a = 3;
4      x = 2./a;
5      float x;
6      printf("a=%d x=%f\n", a, x);
7      return 0;
8  }
```

Code erroné

Est-ce que l'utilisation de a et de x est conforme à leur portée ?
non !

```
1  int main()
2  {
3      int a = 3;          <-- debut portee de a
4      x = 2./a;           <-- on peut utiliser a mais pas x
5      float x;            <-- debut portee de x
6      printf("a=%d x=%f\n", a, x);
7      return 0;
8  }
```

Le compilateur affiche alors le message suivant :
error : x was not declared in this scope

Exemple 2

Quelles sont les portées de s, i et p ?

```
1  int main() {
2      int s = 0;
3      for (int i=0;i<10;i++) {
4          int p;
5          p = i*i;
6          s = s + p ;
7      }
8      printf("S = %d\n",s);
9      return 0;
10 }
```

Exemple 2

Quelles sont les portées de s, i et p ?

```
1  int main() {
2      int s = 0;                          <-- debut portee de s
3      for (int i=0;i<10;i++) {            <-- debut portee de i
4          int p;                          <-- debut portee de p
5          p = i*i;
6          s = s + p;
7      }                                   <-- fin portee de p et i
8      printf("S = %d\n",s);
9      return 0;
10 }                                       <-- fin portee s
```

Résolution des noms

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Résolution des noms

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Réponse

✗ Impossible si les variables sont définies dans le même bloc

✓ Possible si les variables sont définies dans des blocs différents
↪ même s'ils sont imbriqués
⇒ la variable utilisée est celle du bloc englobant le plus proche

Résolution des noms

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Réponse

✗ Impossible si les variables sont définies dans le même bloc

✓ Possible si les variables sont définies dans des blocs différents
↪ même s'ils sont imbriqués
⇒ la variable utilisée est celle du bloc englobant le plus proche

```c
int main() {
    int a = 3;
    {
        int a = 5;
        printf("a = %d \n", a);
    }
    return 0;
}
```

Qu'affiche ce programme ?

Résolution des noms

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Réponse

✗ Impossible si les variables sont définies dans le même bloc

✓ Possible si les variables sont définies dans des blocs différents
↪ même s'ils sont imbriqués
⇒ la variable utilisée est celle du bloc englobant le plus proche

```c
int main() {
    int a = 3;
    {
        int a = 5;
        printf("a = %d \n", a);
    }
    return 0;
}
```

Qu'affiche ce programme ?

Le programme affiche : a = 5

Même nom dans des fonctions différentes

Question

Un problème ?

```
1  float cube(float x) {
2      float c = x*x*x;
3      return c;
4  }
5
6  int main() {
7      float c;
8      c = cube(2);
9      printf("le cube de 2 est %f\n", c);
10     return 0;
11 }
```

Même nom dans des fonctions différentes

Question

Un problème ?

```
1  float cube(float x) {
2      float c = x*x*x*x;
3      return c;
4  }
5
6  int main() {
7      float c;
8      c = cube(2);
9      printf("le cube de 2 est %f\n", c);
10     return 0;
11 }
```

Règles de portée liées au bloc $\Rightarrow$ pas de problème

on a ici deux variables différentes de nom c qui ne sont accessibles qu'au sein du bloc de la définition de leur fonction.

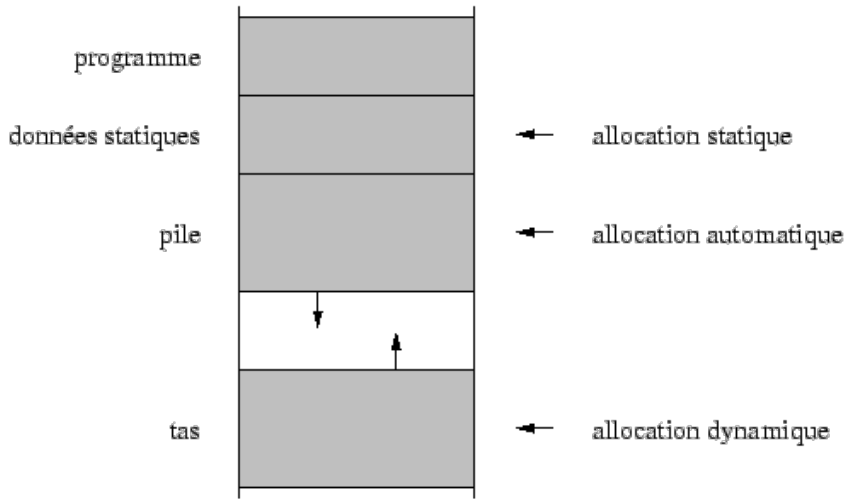
Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Variables temporaires

Les **variables locales (classiques)** que nous avons utilisées jusqu'alors sont des variables temporaires :

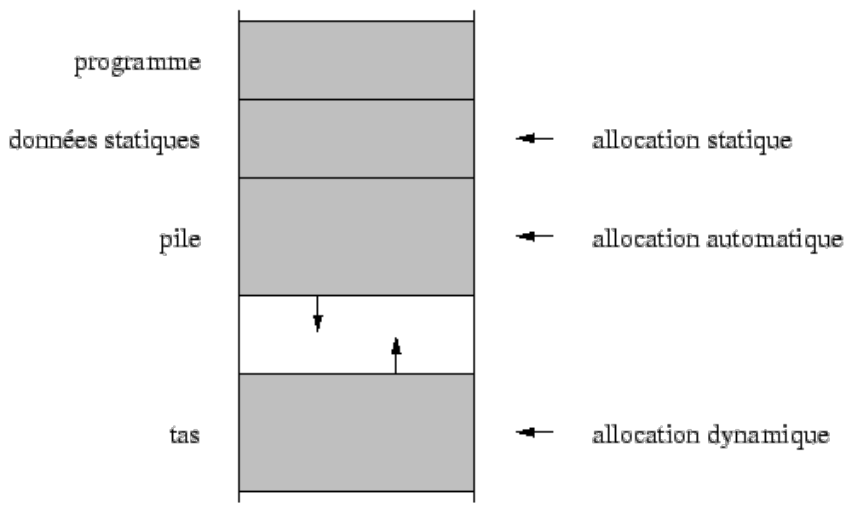
- elles n'existent que durant un appel à la fonction dans laquelle elle sont définies
- un nouvel exemplaire est recréé à chaque appel et détruit au retour de fonction
- un emplacement est alloué dans la pile pour chaque exemplaire : on parle d'**allocation automatique**.



Variables permanentes

Les variables permanentes existent durant toute l'exécution du programme :

- un emplacement leur est alloué dans le segment de données durant la compilation : on parle d'**allocation statique**.
- on distingue deux classes de variables permanentes :
  - les **variables locales permanentes** déclarées dans une fonction mais dont le type est précédé du mot-clé `static`
  - les **variables globales** déclarées en dehors de toute fonction



Variables locales permanentes

Définition

Une variable locale permanente est une variable **déclarée dans une fonction** par : `static type nom;`

- Les mêmes règles de portée s'appliquent que pour les variables locales temporaires (classiques) ;
- Tous les appels à la fonction partagent le même exemplaire de la variable et **sa valeur est conservée** d'un appel à l'autre
- Lors de sa création elle est **initialisée à 0** quelque soit son type (une valeur initiale littérale peut lui être donnée).

Variables locales permanentes : exemple

```c
#include <stdio.h>

int prochPuis() {
    static int id ;
    if (id) id = id * 2;
    else id = 1;
    return id ;
}

int main() {
    for(int i = 1; i<=10 ;i++)
        printf("prochaine puissance : %d\n",prochPuis());
    return 0;
}
```

Variables locales permanentes : exemple

```c
#include <stdio.h>

int prochPuis() {
    static int id ;
    if (id) id = id * 2;
    else id = 1;
    return id ;
}

int main() {
    for(int i = 1; i<=10 ;i++)
        printf("prochaine puissance : %d\n",prochPuis());
    return 0;
}
```

Exécution :

```
prochaine puissance : 1
prochaine puissance : 2
prochaine puissance : 4
prochaine puissance : 8
prochaine puissance : 16
prochaine puissance : 32
prochaine puissance : 64
prochaine puissance : 128
prochaine puissance : 256
prochaine puissance : 512
```

Variables globales

Définition

Une variable globale est une variable **déclarée en dehors de toute fonction** :

- Les règles de portée changent : une variable globale est connue dans toutes les **fonctions situées après sa déclaration** dans le fichier.
  - cependant si une variable locale (ou un paramètre) porte le même nom qu'une variable globale, elle "masque" la variable globale qui n'est plus accessible dans cette fonction
- Une variable globale existe en un seul exemplaire
- Lors de sa création elle est **initialisée à 0** (une valeur initiale littérale peut lui être donnée).

Portée des variables globales : exemple

```c
void fonction1() {
  // ne peut pas utiliser la variable globale a
}

int a = 3;

void fonction2() {
  // peut utiliser la variable globale a
}

void fonction3(float a) {
  // ne peut utiliser la variable globale a, masquee par le parametre
}

void fonction4() {
  int a;
  // ne peut utiliser la variable globale a, masquee par la variable locale
}

int main() {
  // peut utiliser la variable globale a
    return 0;
}
```

Variables globales : c'est dangereux !

Qu'affiche le code suivant :

```
1  int i;
2
3  void coucou() {
4      for (i=0;i<3;i++)
5          printf("coucou %d fois \n", i);
6  }
7
8  int main() {
9      for (i=0;i<3;i++)
10         coucou();
11     printf("fin\n");
12     return 0;
13 }
```

Variables globales : c'est dangereux !

Qu'affiche le code suivant :

```
1  int i;
2
3  void coucou() {
4      for (i=0;i<3;i++)
5          printf("coucou %d fois \n", i);
6  }
7
8  int main() {
9      for (i=0;i<3;i++)
10         coucou();
11     printf("fin\n");
12     return 0;
13 }
```

coucou 0 fois
coucou 1 fois
coucou 2 fois
fin

Pourquoi c'est dangereux !

Remarque

Variable globale un moyen de transmission de valeurs entre fonctions (plus de paramètre) ?

✓ Si une variable globale permet d'économiser de la place mémoire et du temps de transmission de valeur elle impose au programmeur de **gérer la cohérence de son utilisation**.

✗ On perd l'intérêt de la programmation structurée qui permet de limiter aux paramètres et résultat les interactions avec les autres fonctions.

Conclusion :

- Une variable globale n'est pertinente que si sa valeur ne change jamais $\Rightarrow$ n'utilisons que des **variables globales constantes**.

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Les différentes formes de constante

Une constante peut être représentée en C par :

- une simple **valeur littérale** donnée dans le source du programme
- une **macro** définie par la directive de pré-compilation `#define`
- une **variable globale** déclarée avec le mot-clé `const`

Les deux dernières formes permettent de **nommer** la constante :

- Cela incarne la constante indépendamment de sa valeur et assure la cohérence de son utilisation ;
- On peut modifier à un seul endroit du code la valeur de la constante.

La variable globale constante permet en plus de **typer** la constante indépendamment de sa valeur.

La macro constante

Syntaxe

```
#define NOM littéral
```

Exemple

```
#define MAX 1000
#define PI 3.1415
```

Remarque : par convention, les macros sont notées en majuscules.

Exemple

Le pré-processeur remplace toutes les occurrences des macro par leur littéral associé.

```c
#include <stdio.h>

#define MAX 10

void produit(int a) {
    int i;
    for (i=1;i<=MAX;i++)
        printf("%d ", a*i);
}
int main() {
    int i;
    for (i=1;i<=MAX; i++){
      produit(i);
      printf("\n");
    }
    return 0;
}
```

```c
// contenu de stdio.h



void produit(int a) {
    int i;
    for (i=1;i<=10;i++)
        printf("%d ", a*i);
}
int main() {
    int i;
    for (i=1;i<=10; i++){
      produit(i);
      printf("\n");
    }
    return 0;
}
```

La variable globale constante

Syntaxe

```
const type nom = littéral;
```

Exemple

```
const int max = 10;
const double pi = 3.1415;
```

Exemple

```c
#include <stdio.h>

const int max = 10;

void produit(int a) {
    int i;
    for (i=1;i<=max;i++)
        printf("%d ", a*i);
}

int main() {
    int i;
    for (i=1;i<=max; i++){
      produit(i);
      printf("\n");
    }
    return 0;
}
```

Protection de la variable globale constante

Le compilateur interdit toute affectation d'une variable déclarée avec le mot clé `const`.

```
1 const int max = 10;
2 int main() {
3     max = 11;
4     return 0;
5 }
```

Le compilateur affiche : *error : assignment of read-only variable 'max'*

Remarque

Le mot-clé `const` peut aussi être mis sur des paramètres pour bloquer la possibilité de les modifier, ou sur des variables locales pour en faire des constantes locales.

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Ordre des fonctions dans le source

Question

Quand on définit une fonction, à quel endroit peut-on l'utiliser ?

Réponse

Une fonction est utilisable dès que l'on a précisé sa signature
$\Rightarrow$ donc partout après sa **définition**

Contraintes

- Si une fonction `f1` appelle une fonction `f2`, la définition de `f2` doit se situer avant la fonction `f1`
- La fonction `main` doit être située à la fin du fichier source.

Exemple

L'ordre de définition des fonctions contraint les appels réalisables.

```
1  void f1() {
2      // je ne peux pas me servir de f2
3  }
4
5  void f2() {
6    f1(); // je peux me servir de f1
7  }
8
9  int main() {
10     f1(); // je peux me servir de f1
11     f2(); // je peux me servir de f2
12 }
```

Exemple

L'ordre de définition des fonctions contraint les appels réalisables.

```
1  void f1() {
2      // je ne peux pas me servir de f2
3  }
4
5  void f2() {
6    f1(); // je peux me servir de f1
7  }
8
9  int main() {
10     f1(); // je peux me servir de f1
11     f2(); // je peux me servir de f2
12 }
```

Plutôt : l'ordre de définition des fonctions doit se plier aux appels que l'on souhaite faire entre fonctions.

Le cas des fonctions récursives

Définition

Une fonction est récursive si elle s'appelle elle-même.

Mise en œuvre implicite en C

Rendu possible car une fonction est utilisable dès qu'on a spécifié sa signature, ce qui est le cas dans le bloc de définition d'une fonction (qui vient forcément après sa signature).

Exemple

Somme des carrés des entiers compris entre m et n :

$$sommeCarrés(m,n) = \begin{cases} 0 & \text{si } m > n \\ m^2 + sommeCarrés(m+1,n) & \text{sinon} \end{cases}$$

```
1  int sommeCarres(int m, int n) {
2      if (m>n) return 0;
3      else return (m*m + sommeCarres(m+1, n));
4  }
5
6  int main() {
7      int m = 5, n = 10;
8      printf("Carres entre 5 et 10 : %d\n", sommeCarres(5,10));
9      return 0;
10 }
```

Exemple

Somme des carrés des entiers compris entre m et n :

$$sommeCarrés(m,n) = \begin{cases} 0 & \text{si } m > n \\ m^2 + sommeCarrés(m+1,n) & \text{sinon} \end{cases}$$

```
1  int sommeCarres(int m, int n) {
2      if (m>n) return 0;
3      else return (m*m + sommeCarres(m+1, n));
4  }
5
6  int main() {
7      int m = 5, n = 10;
8      printf("Carres entre 5 et 10 : %d\n", sommeCarres(5,10));
9      return 0;
10 }
```

Remarque : on peut aussi en donner une vision purement fonctionnelle :

```
int sommeCarres(int m, int n) {
    return (m>n ? 0 : m*m+sommeCarres(m+1,n)); }
```

Problème : les fonctions inter-dépendantes

On souhaite définir ces deux fonctions :

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{sinon} \end{cases}$$

Quelle est la fonction qui se place avant l'autre ?

Problème : les fonctions inter-dépendantes

On souhaite définir ces deux fonctions :

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{sinon} \end{cases}$$

Quelle est la fonction qui se place avant l'autre ?

Aucune, elles sont inter-dépendantes ! ! !

Problème : les fonctions inter-dépendantes

Solution

On peut **déclarer** une fonction, c'est-à-dire donner sa signature, sans la **définir**.

Intérêt : je peux faire un appel à cette fonction dans n'importe quelle fonction définie après cette déclaration sans avoir à préciser sa définition.

Définition

La déclaration d'une fonction est simplement composée de la signature de la fonction suivie d'un `;`.

Syntaxe

```
typeRes nomFonction(type1, type2, ...);
```

Remarque : Les noms des paramètres ne sont pas obligatoires

Retour à Pair/Impair

```
1  _Bool impair(int);
2
3  _Bool  pair(int n) {
4      if (n==0) return true;
5      else      return impair(n-1);
6  }
7
8  _Bool impair(int n) {
9      if (n==0) return false;
10     else      return pair(n-1);
11 }
```

Plan

1 Fonctions
- Introduction
- Définition d'une fonction
- Appel d'une fonction
- Portée des variables
- Variables permanentes
- Constantes
- Portée des fonctions
- Programmation modulaire

Structure classique d'un fichier source

```
1  /* Debut declarations de fonctions */
2  _Bool pair(int);
3  _Bool impair(int);
4  ...
5  /* Fin declarations de fonctions */
6
7  /* Debut definitions de fonctions */
8  _Bool pair(int n){
9      if (n==0) return true;
10     else return impair(n-1);
11 }
12 ...
13 /* Fin definitions de fonctions */
14
15 int main() {
16     ....
17     return 0;
18 }
```

Modules

Remarque

Il est possible de scinder son programme (l'ensemble des fonctions qui le composent) en plusieurs fichiers : on parle de modules.

- Les **modules** sont des regroupements cohérents de fonctions.
- Les librairies du `C` sont un exemple de modules :
  - `stdio` dédiée aux entrées-sorties
  - `math` dédiée aux fonctions mathématiques
  - `string` dédiée aux fonctions sur les chaînes de caractères
  - ...
- L'utilisation de modules :
  - évite de manipuler des fichiers sources trop grands,
  - facilite la réutilisation de portions de code,
  - facilite la partage du travail sur une équipe de programmeur,
  - facilite l'amélioration des différentes parties du code

Structure d'un module

Chaque module `module` est composé de deux fichiers :

- un **fichier d'en-tête** `module.h` spécifiant les fonctions et constantes "offertes" par le module,
- un **fichier programme** `module.c` contenant les définitions des fonctions déclarées dans `module.h` et éventuellement d'autres fonctions et constantes utiles au module mais non accessibles en dehors du module.

D'un point de vue utilisateur du module, le module est une boite noire dont la seule connaissance est l'ensemble des déclarations du fichier `module.h`. L'utilisateur aura à inclure ce fichier pour utiliser le module : `#include "module.h"`.
D'un point de vue concepteur du module, le module est une boite blanche dont il spécifie l'accès dans le fichier `module.h` et dont il définit le fonctionnement dans le fichier `module.c`. Le fichier `module.c` doit inclure le fichier `module.h` pour assurer la cohérence du module.

Exemple

pairImpair.h

```
_Bool pair(int n);
_Bool impair(int n);
```

pairImpair.c

```
#include "pairImpair.h"

_Bool  pair(int n) {
    if (n==0) return true;
    else return impair(n-1);
}

_Bool impair(int n){
    if (n==0) return false;
    else return pair(n-1);
}
```

prog.c

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

int main() {
    int i = 0;
    printf("Entrez un entier positif\n");
    scanf("%d",&i);
    if (i>0) {
        if (pair(i)) printf("%d est pair\n",i);
        else printf("%d est impair\n",i);
    }
    else printf("%d n'est pas positif\n",i);
    return 0;
}
```

Problème d'inclusions multiples

Si le fichier d'en-tête est inclus plusieurs fois, cela provoque une erreur de compilation (redéclaration de fonction). Pour éviter cela, on utilise des directives du pré-processeur pour n'inclure les déclarations d'un module qu'une seule fois.

pairImpair.h

```
1 #ifndef PAIRIMPAIR_H
2 #define PAIRIMPAIR_H
3
4 _Bool pair(int);
5 _Bool impair(int);
6
7 #endif
```

Spécifications des fonctions

Afin de permettre une utilisation informée des fonctions d'un module, il est **indispensable de précisément décrire les spécifications** de chacune des fonctions offertes par un module.

- le résumé du rôle de la fonction
- la description du rôle de chacun des paramètres
- la description du résultat en fonction des paramètres
- toute pré-condition garantissant son bon fonctionnement

Ces commentaires peuvent être mis directement `/* ... */` ou suivre un format pré-defini permettant une génération automatique d'une documentation du module. Ici les commentaires sont donnés au format `doxygen`.

pairImpair.h

```
/**
\brief teste si un entier est pair
\param n est un entier naturel
\return 1 si n est pair, 0 sinon
*/
_Bool pair(int n);
```

Algorithmes et fonctions C ?

Comme en algorithmique, une fonction C comporte :

- **un nom**,
- **des paramètres nommés et typés**,
- **un résultat typé**,
- **des variables locales**,
- **une description** du calcul effectué par la fonction.

A la différence de l'algorithmique :

- le **rôle** (E, S, ES) des paramètres n'est pas précisé. Ils ont tous le **rôle Entrée**.
- pas de distinction **fonction**/**procédure** mais une fonction C peut ne pas renvoyer de valeur (le type de son résultat est `void`)

Question

Comment implanter des algorithmes ayant des paramètres **Sortie** ou **Entrée-Sortie** ?