Fonctions

A. Où en est-on ?

1. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?

A. Où en est-on ?

1. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?
2. Qui a pu utiliser notre serveur JupyterHub depuis « chez lui »

A. Où en est-on ?

1. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?
2. Qui a pu utiliser notre serveur JupyterHub depuis « chez lui »
3. Qui a fini les TD et TP de la semaine dernière ?

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. reformulé par soi-même : 80 %

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. reformulé par soi-même : 80 %

Mémorisation sur le long terme

- Cours relu le soir même : 80 %
- Cours relu plus tard : 10 %

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. reformulé par soi-même : 80 %

Mémorisation sur le long terme

- Cours relu le soir même : 80 %
- Cours relu plus tard : 10 %

Objectif : **Travailler efficacement**

- Voir la page web pour des recommandations

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :

- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :

- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :

- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Remarque

Tout ce qui est calculable par un ordinateur peut être programmé uniquement avec ces instructions

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :

- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Remarque

Tout ce qui est calculable par un ordinateur peut être programmé uniquement avec ces instructions
(ou presque : il faudrait un accès un peu plus souple à la mémoire)

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :

- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Remarque

Tout ce qui est calculable par un ordinateur peut être programmé uniquement avec ces instructions
(ou presque : il faudrait un accès un peu plus souple à la mémoire)

Pourquoi aller plus loin ?

Résumé des épisodes précédents . . .

Pour le moment nous avons vu :
- Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
- Variables, types, affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Remarque

Tout ce qui est calculable par un ordinateur peut être programmé uniquement avec ces instructions
(ou presque : il faudrait un accès un peu plus souple à la mémoire)

Pourquoi aller plus loin ?

Passage à l'échelle !

Motivation : l'exemple du livre de cuisine (1)

Recette de la tarte aux pommes

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 100 g de sucre en poudre, 5 belles pommes
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau
- Mettre le beurre
- Mettre le sucre et le sel
- Pétrir de façon à former une boule
- Étaler la pâte dans un moule
- Peler les pommes, les couper en quartiers et les disposer sur la pâte
- Faire cuire 30 minutes

Motivation : l'exemple du livre de cuisine (2)

Recette de la tarte aux poires

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 100 g de sucre en poudre, 5 belles poires
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau
- Mettre le beurre
- Mettre le sucre et le sel
- Pétrir de façon à former une boule
- Étaler la pâte dans un moule
- Peler les poires, les couper en quartiers et les disposer sur la pâte
- Faire cuire 30 minutes

Motivation : l'exemple du livre de cuisine (3)

Recette de la tarte tatin

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 200 g de sucre en poudre, 5 belles pommes
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau
- Mettre le beurre
- Mettre le sucre et le sel
- Pétrir de façon à former une boule
- Verser le sucre dans une casserole
- Rajouter un peu d'eau pour l'humecter
- Le faire caraméliser à feu vif, sans remuer
- Verser au fond du plat à tarte
- Peler les pommes, les couper en quartiers
- Faire revenir les pommes dans une poêle avec du beurre
- Disposer les pommes dans le plat et étaler la pâte au dessus
- Faire cuire 45 minutes et retourner dans une assiette

Qu'est-ce qui ne va pas ?

Qu'est-ce qui ne va pas ?

Duplication

- Longueurs
- En cas d'erreur ou d'amélioration : corriger plusieurs endroits !

Qu'est-ce qui ne va pas ?

Duplication

- Longueurs
- En cas d'erreur ou d'amélioration : corriger plusieurs endroits !

Manque d'expressivité

- Difficile à lire
- Difficile à mémoriser

Qu'est-ce qui ne va pas ?

Duplication

- Longueurs
- En cas d'erreur ou d'amélioration : corriger plusieurs endroits !

Manque d'expressivité

- Difficile à lire
- Difficile à mémoriser

Essayons d'améliorer cela

Recettes de base

Recette de la pâte brisée

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau Mettre le beurre
- Mettre le sucre et et une pincée de sel
- Pétrir de façon à former une boule

Recette du caramel

- Ingrédients : 100 g de sucre
- Verser le sucre dans une casserole
- Rajouter un peu d'eau pour l'humecter
- Le faire caraméliser à feu vif, sans remuer

Recettes de tartes

Tarte aux fruits (pommes, poires, ...)

- Ingrédients : 500g de fruits, ingrédients pour une pâte brisée
- **Préparer une pâte brisée**
- Étaler la pâte dans un moule
- Peler les fruits, les couper en quartiers et les disposer sur la pâte
- Faire cuire 30 minutes

Recettes de tartes

Tarte aux fruits (pommes, poires, ...)

- Ingrédients : 500g de fruits, ingrédients pour une pâte brisée
- **Préparer une pâte brisée**
- Étaler la pâte dans un moule
- Peler les fruits, les couper en quartiers et les disposer sur la pâte
- Faire cuire 30 minutes

Tarte tatin

- Ingrédients : 5 belles pommes, pâte brisée, caramel
- **Préparer une pâte brisée**
- **Préparer un caramel** et le verser au fond du plat à tarte
- Peler les pommes, les couper en quartiers
- Faire revenir les pommes dans une poêle avec du beurre
- Disposer les pommes dans le plat, et étaler la pâte au dessus
- Faire cuire 45 minutes et retourner dans une assiette

Les **fonctions** : objectif

Modularité

– Décomposer un programme en programmes plus simples

Les **fonctions** : objectif

Modularité

- Décomposer un programme en programmes plus simples
- Implantation plus facile
- Validation (tests)
- Réutilisation
- Flexibilité (remplacement d'un sous-programme par un autre)

Les **fonctions** : objectif

Modularité

- Décomposer un programme en programmes plus simples
- Implantation plus facile
- Validation (tests)
- Réutilisation
- Flexibilité (remplacement d'un sous-programme par un autre)

Non duplication

- Partager (**factoriser**) du code
- Code plus court
- Maintenance plus facile

Les **fonctions** : objectif

Modularité

- Décomposer un programme en programmes plus simples
- Implantation plus facile
- Validation (tests)
- Réutilisation
- Flexibilité (remplacement d'un sous-programme par un autre)

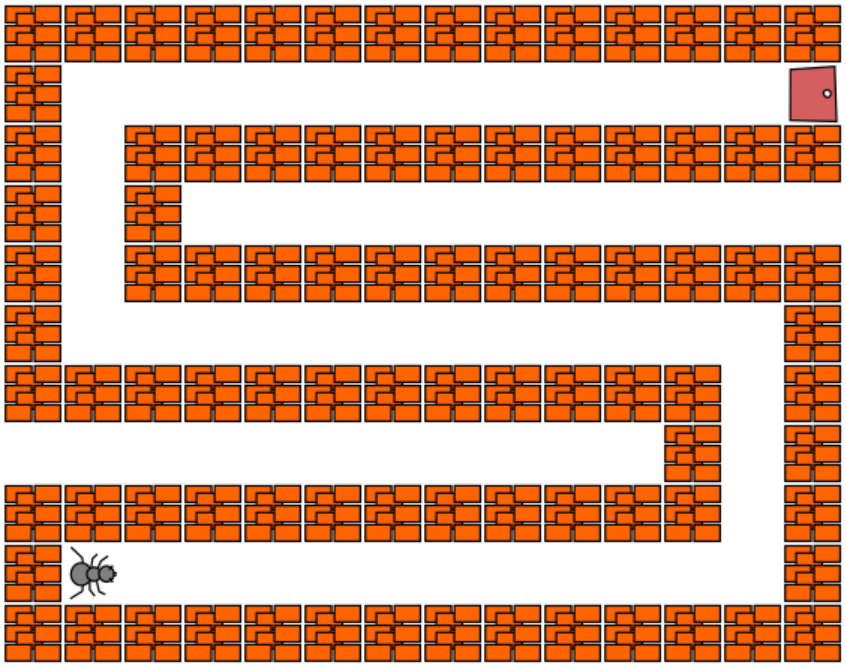
Non duplication

- Partager (**factoriser**) du code
- Code plus court
- Maintenance plus facile

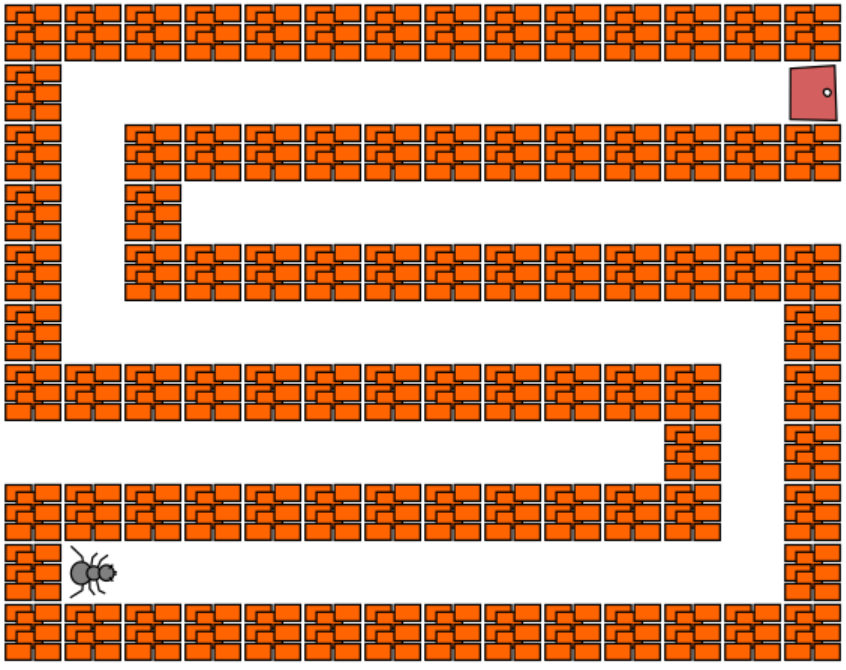
Niveau d'abstraction

- Programmes plus **concis** et **expressifs**

Une impression de déjà vu ?



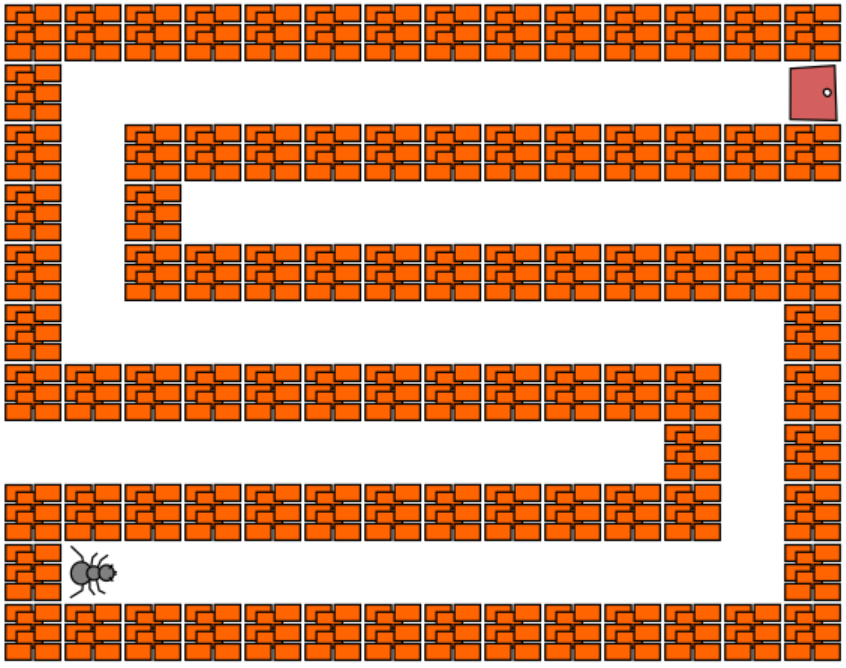
Une impression de déjà vu ?



laby2c-mauvais.cpp

```
void fourmi() {
    while ( regarde() == Vide ) {
        avance();
    }
    gauche();
    while ( regarde() == Vide ) {
        avance();
    }
    gauche();
    while ( regarde() == Vide ) {
        avance();
    }
    droite();
    while ( regarde() == Vide ) {
        avance();
    }
    droite();
    while ( regarde() == Vide ) {
        avance();
    }
    ouvre();
}
```

Une impression de déjà vu ?



laby2c.cpp

```
void avanceTantQueTuPeux() {
    while ( regarde() == Vide ) {
        avance();
    }
}

void fourmi() {
    avanceTantQueTuPeux();
    gauche();
    avanceTantQueTuPeux();
    gauche();
    avanceTantQueTuPeux();
    droite();
    avanceTantQueTuPeux();
    droite();
    avanceTantQueTuPeux();
    ouvre();
}
```

Une impression de déjà vu ?

Fonctions que vous avez déjà écrites

- TD1 : `transporte(Chèvre)`

Une impression de déjà vu ?

Fonctions que vous avez déjà écrites

- TD1 : `transporte(Chèvre)`
- TP1 : `avanceTantQueTuPeux()`

Une impression de déjà vu ?

Fonctions que vous avez déjà écrites

- TD1 : `transporte(Chèvre)`
- TP1 : `avanceTantQueTuPeux()`
- TD2 : `max(note1, note2)`, `max(note1, note2, note3)`

Une impression de déjà vu ?

Fonctions que vous avez déjà écrites

- TD1 : `transporte(Chèvre)`
- TP1 : `avanceTantQueTuPeux()`
- TD2 : `max(note1, note2)`, `max(note1, note2, note3)`
- TP3 : `factorielle(n)`

Appel de fonctions usuelles

fonctions-usuelles.ipynb

Chargement de la bibliothèque de fonctions mathématiques usuelles:

```
In [1]: #include <cmath>
```

Fonctions trigonométriques:

```
In [2]: cos(3.14159)
```

```
Out[2]: -1
```

Fonction exponnentielle:

```
In [3]: exp(1.0)
```

```
Out[3]: 2.71828
```

Fonction puissance:

```
In [4]: pow(3, 2)
```

```
Out[4]: 9
```

```
In [5]: pow(2, 3)
```

```
Out[5]: 8
```

Appel de fonctions usuelles

On remarque

– La présence de `#include <cmath>`
  C'est pour utiliser la bibliothèque de fonctions mathématiques
  On y reviendra ...

Appel de fonctions usuelles

On remarque

- La présence de `#include <cmath>`
  C'est pour utiliser la bibliothèque de fonctions mathématiques
  On y reviendra ...

- L'ordre des arguments est important
- Le type des arguments est important

Appel de fonctions usuelles

On remarque

- La présence de `#include <cmath>`
  C'est pour utiliser la bibliothèque de fonctions mathématiques
  On y reviendra ...

- L'ordre des arguments est important
- Le type des arguments est important

- On sait ce que calcule `cos(x)` !
- On ne sait pas **comment** il le fait
- **On n'a pas besoin de le savoir**

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.ipynb

La fonction factorielle

À la main

Calculons 5!

```
In [1]: int resultat;
```

```
In [2]: resultat = 1;
        for ( int i = 1; i <= 5; i++ ) {
            resultat = resultat * i;
        }
        resultat
```

```
Out[2]: 120
```

Calculons 7!

```
In [3]: resultat = 1;
        for ( int i = 1; i <= 7; i++ ) {
            resultat = resultat * i;
        }
        resultat
```

```
Out[3]: 5040
```

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.ipynb

Avec une fonction

```
In [4]: int factorielle(int n) {
            int resultat = 1;
            for ( int i = 1; i <= n; i++ ) {
                resultat = resultat * i;
            }
            return resultat;
        }

In [5]: factorielle(5)

Out[5]: 120

In [6]: factorielle(7)

Out[6]: 5040

In [7]: factorielle(5) / factorielle(3) / factorielle(2)

Out[7]: 10
```

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.cpp

```
int factorielle(int n) {
    int resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
    return resultat;
}
```

Syntaxe d'une fonction

Syntaxe

```
type nom(type1 parametre1, type2 parametre2, ...) {
    déclarations de variables;
    bloc d'instructions;
    return expression;
}
```

- `parametre1, parametre2, ...` : les *paramètres formels*
- Le type des paramètres formels est fixé
- Les variables sont appelées *variables locales*
- À la fin, la fonction *renvoie* la valeur de `expression`
  Celle-ci doit être du type annoncé

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

– `max` n'est utilisée que si elle est **appelée**

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

– `max` n'est utilisée que si elle est **appelée**
– Pour appeler cette fonction on écrit, par exemple,

```
max(1.5, 3.0)
```

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

- `max` n'est utilisée que si elle est **appelée**
- Pour appeler cette fonction on écrit, par exemple,

```
max(1.5, 3.0)
```

- les *paramètres* `a` et `b` sont initialisés avec les valeurs 1.5 et 3.0

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

- `max` n'est utilisée que si elle est **appelée**
- Pour appeler cette fonction on écrit, par exemple,

```
max(1.5, 3.0)
```

- les *paramètres* `a` et `b` sont initialisés avec les valeurs 1.5 et 3.0
- le code de la fonction est exécuté

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

- `max` n'est utilisée que si elle est **appelée**
- Pour appeler cette fonction on écrit, par exemple,

```
max(1.5, 3.0)
```

- les *paramètres* `a` et `b` sont initialisés avec les valeurs 1.5 et 3.0
- le code de la fonction est exécuté
- l'exécution s'arrête au premier `return` rencontré

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

- `max` n'est utilisée que si elle est **appelée**
- Pour appeler cette fonction on écrit, par exemple,

```
max(1.5, 3.0)
```

- les *paramètres* `a` et `b` sont initialisés avec les valeurs 1.5 et 3.0
- le code de la fonction est exécuté
- l'exécution s'arrête au premier `return` rencontré
- le `return` spécifie la *valeur de retour* de la fonction : la valeur de l'expression `max(1.5, 3.0)`.

D. Documentation et tests

Documentation d'une fonction (syntaxe javadoc)

D. Documentation et tests

Documentation d'une fonction (syntaxe javadoc)

Exemple

fonction-factorielle-doctests.cpp

```
/** La fonction factorielle
 *  @param n un nombre entier positif
 *  @return n !
 **/
int factorielle(int n) {
```

D. Documentation et tests

Documentation d'une fonction (syntaxe javadoc)

Exemple

fonction-factorielle-doctests.cpp

```
/** La fonction factorielle
 *  @param n un nombre entier positif
 *  @return n !
 **/
int factorielle(int n) {
```

Une bonne documentation

- Est concise et précise
- Donne les préconditions sur les paramètres
- Décrit le résultat (ce que fait la fonction)

D. Documentation et tests

Documentation d'une fonction (syntaxe javadoc)

Exemple

fonction-factorielle-doctests.cpp

```
/** La fonction factorielle
 *  @param n un nombre entier positif
 *  @return n !
 **/
int factorielle(int n) {
```

Une bonne documentation

- Est concise et précise
- Donne les préconditions sur les paramètres
- Décrit le résultat (ce que fait la fonction)

Astuce pour être efficace

- **Toujours commencer par écrire la documentation**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !

Tests d'une fonction

- Pas d'infrastructure standard en C++ pour écrire des tests
- Dans ce cours, on utilisera une infrastructure minimale

Exemple

fonction-factorielle-doctests.cpp

```
    ASSERT( factorielle(0) == 1  );
    ASSERT( factorielle(1) == 1  );
    ASSERT( factorielle(2) == 2  );
    ASSERT( factorielle(3) == 6  );
    ASSERT( factorielle(4) == 24 );
```

Tests d'une fonction

Astuces pour être efficace

– **Commencer par écrire les tests d'une fonction**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !

Tests d'une fonction

Astuces pour être efficace

- **Commencer par écrire les tests d'une fonction**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !
- Tester les cas particuliers

Tests d'une fonction

Astuces pour être efficace

- **Commencer par écrire les tests d'une fonction**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !
- Tester les cas particuliers
- Tant que l'on est pas sûr que la fonction est correcte :
  - Faire des essais supplémentaires
  - Capitaliser ces essais sous forme de tests

Tests d'une fonction

Astuces pour être efficace

- **Commencer par écrire les tests d'une fonction**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !
- Tester les cas particuliers
- Tant que l'on est pas sûr que la fonction est correcte :
  - Faire des essais supplémentaires
  - Capitaliser ces essais sous forme de tests
- Si l'on trouve un bogue :
  - Ajouter un test caractérisant le bogue

Tests d'une fonction

Astuces pour être efficace

- **Commencer par écrire les tests d'une fonction**
  De toutes les façons il faut réfléchir à ce qu'elle va faire !
- Tester les cas particuliers
- Tant que l'on est pas sûr que la fonction est correcte :
  - Faire des essais supplémentaires
  - Capitaliser ces essais sous forme de tests
- Si l'on trouve un bogue :
  - Ajouter un test caractérisant le bogue

Remarque

- Les effets de bord sont durs à tester !

E. Modèle d'exécution

Objectif

Comprendre précisément l'**appel de fonction**

E. Modèle d'exécution

Objectif

Comprendre précisément l'**appel de fonction**

Exemple : appel de la fonction factorielle

fonction-factorielle.cpp

```
int factorielle(int n) {
    int resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
    return resultat;
}
```

E. Modèle d'exécution

Objectif

Comprendre précisément l'**appel de fonction**

Exemple : appel de la fonction factorielle

fonction-factorielle.cpp

```
int factorielle(int n) {
    int resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
    return resultat;
}
```

Que se passe-t'il lorsque l'on évalue l'expression suivante :

```
factorielle(1+2)
```

Appel de fonctions : formalisation

Syntaxe

nom(expression1, expression2, ...)

Appel de fonctions : formalisation

Syntaxe

nom(expression1, expression2, ...)

Sémantique

1. Evaluation des expressions
2. Leurs valeurs sont les *paramètres réels*

Appel de fonctions : formalisation

Syntaxe

nom(expression1, expression2, ...)

Sémantique

1. Evaluation des expressions
2. Leurs valeurs sont les *paramètres réels*
3. Allocation de mémoire sur la *pile* pour :
   - Les variables locales
   - Les paramètres formels
4. Affectation des paramètres réels aux paramètres formels (par copie ; les types doivent correspondre !)

Appel de fonctions : formalisation

Syntaxe

nom(expression1, expression2, ...)

Sémantique

1. Evaluation des expressions
2. Leurs valeurs sont les *paramètres réels*
3. Allocation de mémoire sur la *pile* pour :
   - Les variables locales
   - Les paramètres formels
4. Affectation des paramètres réels aux paramètres formels (par copie ; les types doivent correspondre !)
5. Exécution des instructions

Appel de fonctions : formalisation

Syntaxe

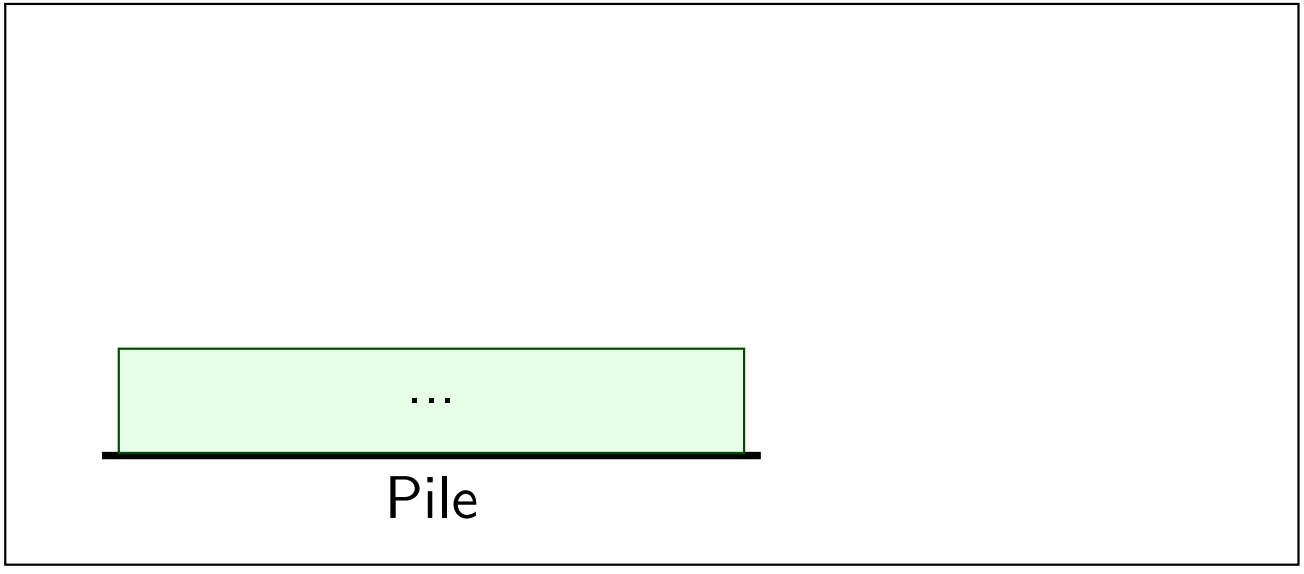
nom(expression1, expression2, ...)

Sémantique

1. Evaluation des expressions
2. Leurs valeurs sont les *paramètres réels*
3. Allocation de mémoire sur la *pile* pour :
   - Les variables locales
   - Les paramètres formels
4. Affectation des paramètres réels aux paramètres formels
   (par copie ; les types doivent correspondre !)
5. Exécution des instructions
6. Lorsque « `return expression` » est rencontré, évaluation de l'expression qui donne la *valeur de retour de la fonction*
7. Désallocation des variables et paramètres sur la pile
8. La valeur de l'expression `nom(...)` est donnée par la valeur de retour

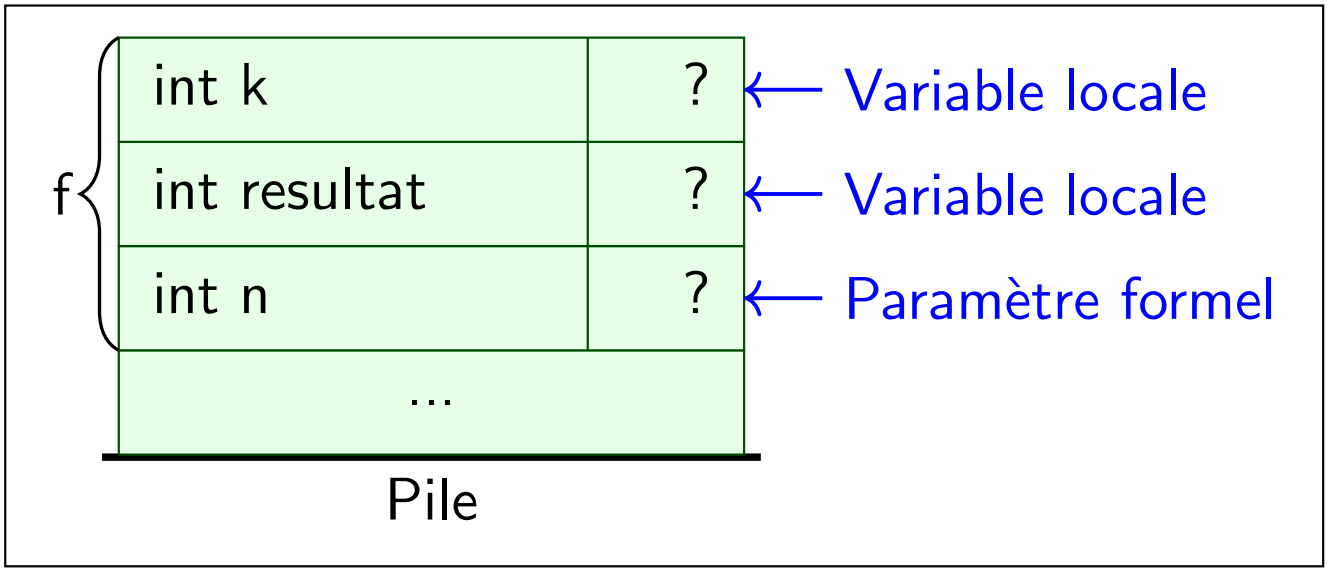
Évolution de la pile sur l'exemple :

1. État initial



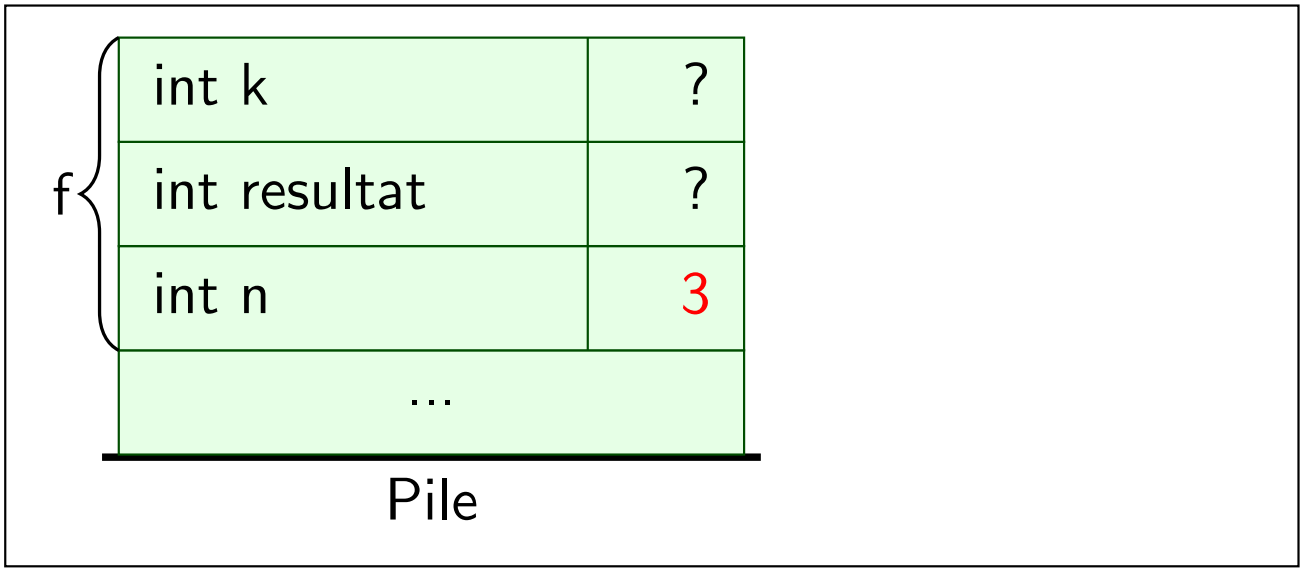
Évolution de la pile sur l'exemple :

3. Allocation de la mémoire sur la pile



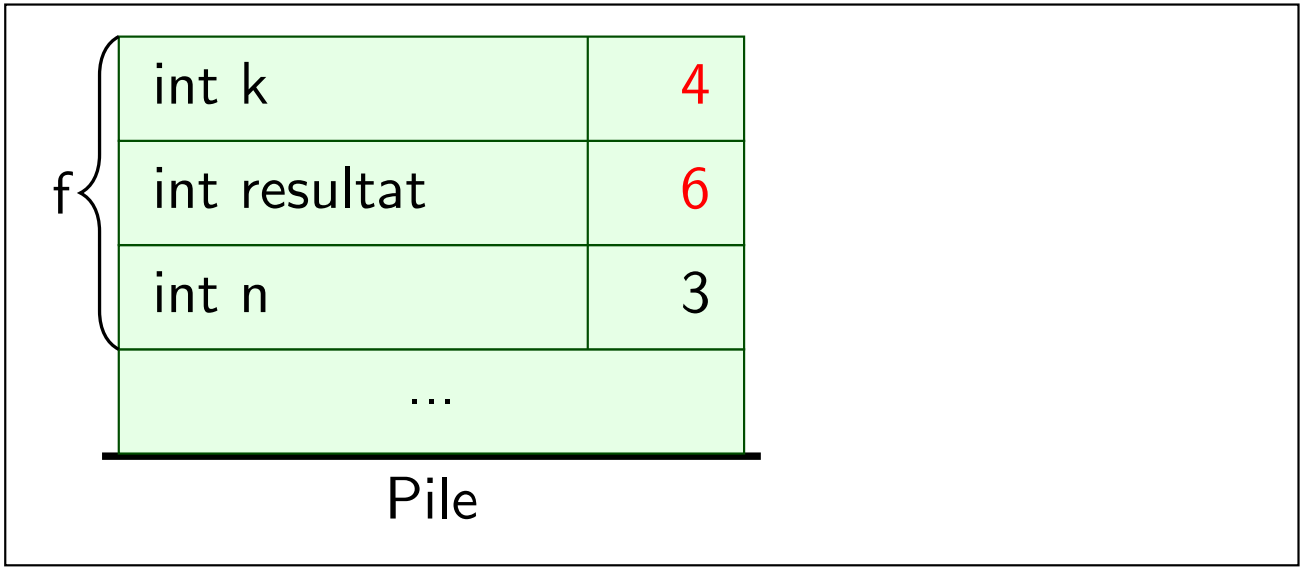
Évolution de la pile sur l'exemple :

4. Affectation du paramètre réel au paramètre formel



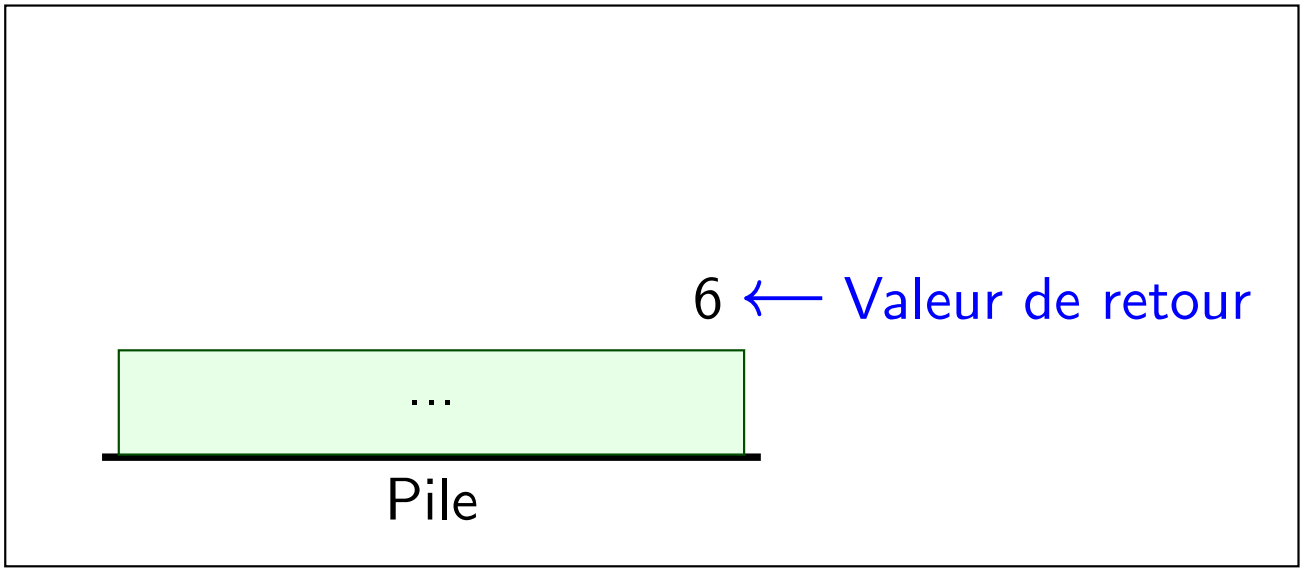
Évolution de la pile sur l'exemple :

5. Exécution des instructions



Évolution de la pile sur l'exemple :

6-8. Désallocation de la mémoire sur la pile et valeur de retour



Appel de fonctions : exercice

On considère la fonction `incremente` :

fonction-exemple-passage-par-valeur.cpp

```
int incremente(int n) {
    n = n + 1;
    return n;
}
```

Quelles sont les valeurs de `a` et `b` après l'exécution des lignes suivantes :

fonction-exemple-passage-par-valeur.cpp

```
    int a, b;

    a = 1;
    b = incremente(a);
```

Passage des paramètres par valeur♣

- Les paramètres formels d'une fonction sont des variables comme les autres
- On peut les modifier
- Mais . . .

Passage des paramètres par valeur♣

- Les paramètres formels d'une fonction sont des variables comme les autres
- On peut les modifier
- Mais . . .

Rappel

Lors d'un appel de fonction ou de procédure, la valeur du paramètre réel est **copiée** dans le paramètre formel

Passage des paramètres par valeur♣

- Les paramètres formels d'une fonction sont des variables comme les autres
- On peut les modifier
- Mais ...

Rappel

Lors d'un appel de fonction ou de procédure, la valeur du paramètre réel est **copiée** dans le paramètre formel

En conséquence

- Une modification du paramètre formel, n'affecte pas le paramètre réel
- Si la variable est volumineuse (tableaux, chaîne de caractères, etc.), cette recopie peut être coûteuse

On dit que les paramètres sont passés *par valeur*
Au second semestre, on verra le passage de paramètres *par référence*

F. Fonctions particulières

1. Procédures
2. Fonctions récursives

Fonctions particulières : Procédures

Besoin de sous-programmes qui **agissent** au lieu de **calculer** :

- on veut produire un effet (affichage, musique, *etc*)
- on veut modifier l'état interne d'une structure de donnée

On parle d'*effet de bord*

Fonctions particulières : Procédures

Besoin de sous-programmes qui **agissent** au lieu de **calculer** :

- on veut produire un effet (affichage, musique, *etc*)
- on veut modifier l'état interne d'une structure de donnée

On parle d'*effet de bord*

Exemple

laby2c.cpp

```
void avanceTantQueTuPeux() {
    while ( regarde() == Vide ) {
        avance();
    }
}
```

Fonctions particulières : Procédures

Besoin de sous-programmes qui **agissent** au lieu de **calculer** :

- on veut produire un effet (affichage, musique, *etc*)
- on veut modifier l'état interne d'une structure de donnée

On parle d'*effet de bord*

Exemple

laby2c.cpp

```
void avanceTantQueTuPeux() {
    while ( regarde() == Vide ) {
        avance();
    }
}
```

Remarques

- Cette fonction ne renvoie rien
- On le dénote en C++ par le type `void`
- Dans d'autres langages on distingue *fonctions* et *procédures*
- Autres exemples : `transporte`, `gauche()`

Fonctions particulières : Fonctions récursives ♣

Exercice

fonction-factorielle-recursive.cpp

```
int factorielle(int n) {
    if ( n == 0 ) {
        return 1;
    } else {
        return n * factorielle(n-1);
    }
}
```

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Fonction

- Combinaison d'instructions élémentaires donnant une instruction de plus haut niveau

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Fonction

- Combinaison d'instructions élémentaires donnant une instruction de plus haut niveau
- Modèle d'exécution (pile)

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Fonction

- Combinaison d'instructions élémentaires donnant une instruction de plus haut niveau
- Modèle d'exécution (pile)
- Fonction main, procédures, fonctions récursives

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Fonction

- Combinaison d'instructions élémentaires donnant une instruction de plus haut niveau
- Modèle d'exécution (pile)
- Fonction main, procédures, fonctions récursives

Trilogie

- **Documentation** : ce que fait la fonction (entrées, sorties, ...)
- **Tests** : ce que fait la fonction (exemples)
- **Code** : comment elle le fait