Fonctions

A. Où en est-on ?

1. Qui a installé `Code::Blocks` et C++ « à la maison » ?

A. Où en est-on ?

1. Qui a installé `Code::Blocks` et `C++` « à la maison » ?
2. Qui a compilé et lancé le programme `bonjour.cpp` « à la maison » ?

A. Où en est-on ?

1. Qui a installé `Code::Blocks` et `C++` « à la maison » ?
2. Qui a compilé et lancé le programme `bonjour.cpp` « à la maison » ?
3. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?

A. Où en est-on ?

1. Qui a installé `Code::Blocks` et `C++` « à la maison » ?
2. Qui a compilé et lancé le programme `bonjour.cpp` « à la maison » ?
3. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?
4. Qui a fini les TD et TP de la semaine dernière ?

A. Où en est-on ?

1. Qui a installé `Code::Blocks` et `C++` « à la maison » ?
2. Qui a compilé et lancé le programme `bonjour.cpp` « à la maison » ?
3. Qui connaît la **syntaxe** et la **sémantique** de la boucle **while** ?
4. Qui a fini les TD et TP de la semaine dernière ?
5. Remarques et suggestions sur le cours, les TD, les TP ?

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. dit : 80 %

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. dit : 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. dit : 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 les instructions suivantes :

- Lecture : `cin >> variable;`
- Écriture : `cout << expression;`

Résumé des épisodes précédents . . .

Pour le moment nous avons vu les instructions suivantes :

- Lecture : `cin >> variable ;`
- Écriture : `cout << expression ;`
- Affectation : `variable = expression`

Résumé des épisodes précédents . . .

Pour le moment nous avons vu les instructions suivantes :

- Lecture : `cin >> variable ;`
- Écriture : `cout << expression ;`
- Affectation : `variable = expression`
- Instruction conditionnelle : `if`
- Instructions itératives : `while`, `do ... while`, `for`

Résumé des épisodes précédents ...

Pour le moment nous avons vu les instructions suivantes :

- Lecture : `cin >> variable;`
- Écriture : `cout << expression;`
- 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 les instructions suivantes :

- Lecture : `cin >> variable ;`
- Écriture : `cout << expression ;`
- 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

Pourquoi aller plus loin ?

Résumé des épisodes précédents . . .

Pour le moment nous avons vu les instructions suivantes :

- Lecture : `cin >> variable;`
- Écriture : `cout << expression;`
- 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

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

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

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 ?

Vous avez déjà écrit des fonctions

- TD1 : `transporter(chèvre)`

Une impression de déjà vu ?

Vous avez déjà écrit des fonctions

- TD1 : `transporter(chèvre)`
- TP1 : `avancer_jusque_au_mur()`

Une impression de déjà vu ?

Vous avez déjà écrit des fonctions

- TD1 : `transporter(chèvre)`
- TP1 : `avancer_jusque_au_mur()`
- TD2 : `max(note1, note2)`, `max(note1, note2, note3)`

Appel de fonctions usuelles

fonctions-usuelles.cpp

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    cout << "cos(pi) = " << cos(3.1415)   << endl;
    cout << "e^1 = "     << exp(1.0)      << endl;
    cout << "3^2 = "     << pow(3.0, 2.0) << endl;
    cout << "2^3 = "     << pow(2.0, 3.0) << endl;
    return 0;
}
```

Appel de fonctions usuelles

fonctions-usuelles.cpp

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    cout << "cos(pi) = " << cos(3.1415)   << endl;
    cout << "e^1 = "     << exp(1.0)      << endl;
    cout << "3^2 = "     << pow(3.0, 2.0) << endl;
    cout << "2^3 = "     << pow(2.0, 3.0) << endl;
    return 0;
}
```

Ce programme affiche :

```
    cos(pi) = -1
    e^1 = 2.71828
    3^2 = 9
    2^3 = 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**

Premiers exemples : la fonction max

max.cpp

```
#include <iostream>
using namespace std;

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

int main() {
    cout << max(1.5, 3.0) << endl;
    cout << max(5.2, 2.0) << endl;
    cout << max(2.3, 2.3) << endl;
    return 0;
}
```

Premiers exemples : la fonction factorielle

factorielle-for.cpp

```
#include <iostream>
using namespace std;

int main() {
    int n, resultat;

    cin >> n;
    resultat = 1;

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

    cout << resultat << endl;

    return 0;
}
```

Premiers exemples : la fonction factorielle

fonction-factorielle.cpp

```
#include <iostream>
using namespace std;

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

int main() {
    int n;
    cin >> n;
    cout << n << "! = " << factorielle(n) << endl;
    return 0;
}
```

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

Confusion **renvoyer** versus **afficher** (exemple)

Que fait le fragment de programme suivant ?

fonction-affichage-bogguee.cpp

```
#include <iostream>
using namespace std;

int maxInt(int a, int b) {
    cin >> a;
    cin >> b;
    if ( a >= b ) {
        cout << a;
    } else {
        cout << b;
    }
}

int main() {
    cout << max(1, max(3, 2)) << endl;
    return 0;
}
```

Confusion **afficher** versus **renvoyer** (analyse)

N'importe quoi !

- la fonction `factorielle` **lit** `n` au clavier
- ... et ignore son paramètre 6

Confusion **afficher** versus **renvoyer** (analyse)

N'importe quoi !

- la fonction `factorielle` **lit** `n` au clavier
- ... et ignore son paramètre 6
- Les résultats intermédiaires sont **affichés**
- ... et perdus pour le programme principal

Confusion **afficher** versus **renvoyer** (analyse)

N'importe quoi !

- la fonction `factorielle` **lit** `n` au clavier
- ... et ignore son paramètre 6
- Les résultats intermédiaires sont **affichés**
- ... et perdus pour le programme principal

Confusions

- **Prendre en paramètre** versus **lire au clavier**
- **Renvoyer** versus **afficher**

Confusion **afficher** versus **renvoyer** (exemple corrigé)

fonction-affichage-correcte.cpp

```
#include <iostream>
using namespace std;

int maxInt(int a, int b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}

int main() {
    cout << maxInt(1, maxInt(3, 2)) << endl;
    return 0;
}
```

Confusion **afficher** versus **renvoyer** (bonnes pratiques)

Une fonction

- **Prend des paramètres** (entrée)
- Effectue un calcul
- **Renvoie** un résultat (sortie)

Confusion **afficher** versus **renvoyer** (bonnes pratiques)

Une fonction

- **Prend des paramètres** (entrée)
- Effectue un calcul
- **Renvoie** un résultat (sortie)

Bonnes pratiques

- Bien séparer :
  - Interaction avec l'utilisateur : lecture et affichage
  - Calcul

Confusion **afficher** versus **renvoyer** (bonnes pratiques)

Une fonction

- **Prend des paramètres** (entrée)
- Effectue un calcul
- **Renvoie** un résultat (sortie)

Bonnes pratiques

- Bien séparer :
  - Interaction avec l'utilisateur : lecture et affichage
  - Calcul
- **Pas de `cin` / `cout` dans une fonction !**
  Exception : procédures ; voir plus loin

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

```
void factorielleTest() {
    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 comment fonctionne l'**appel de fonction** ?

E. Modèle d'exécution

Objectif

Comprendre comment fonctionne 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 comment fonctionne 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

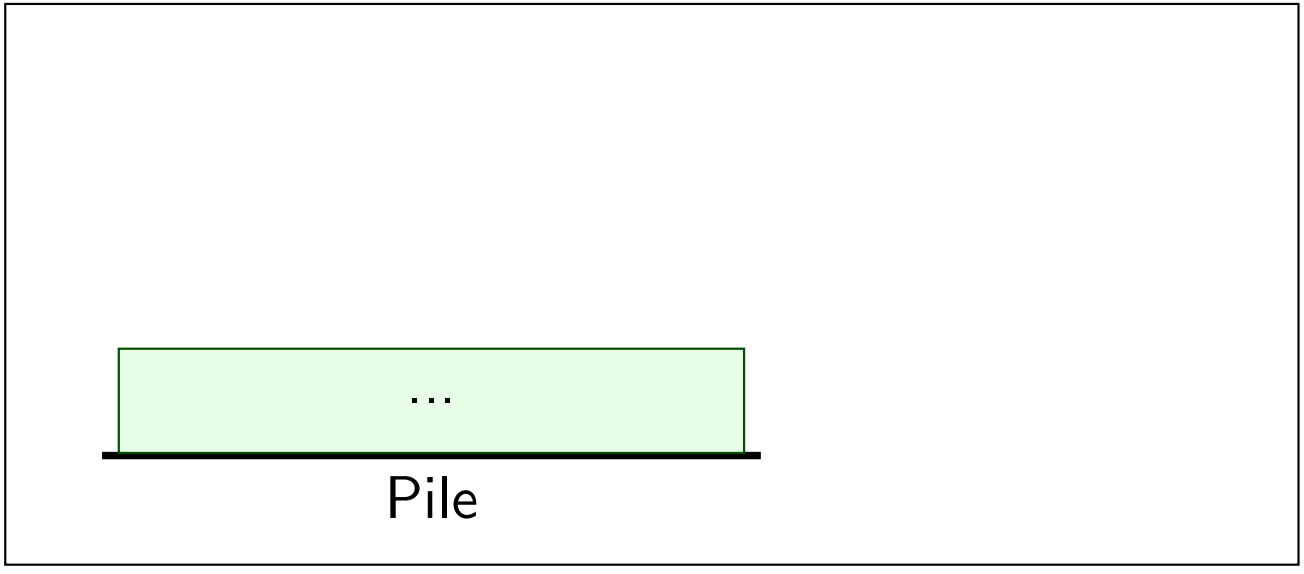
nom(expression1, expression2, ...)

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

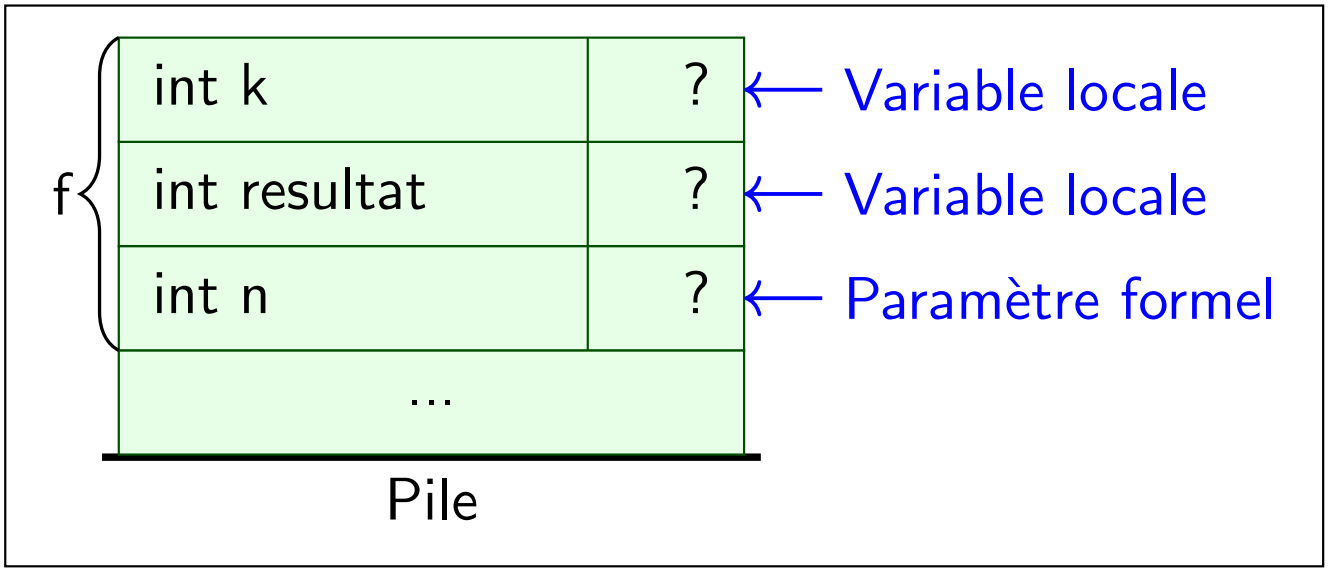
Évolution de la pile sur l'exemple :

1. État initial



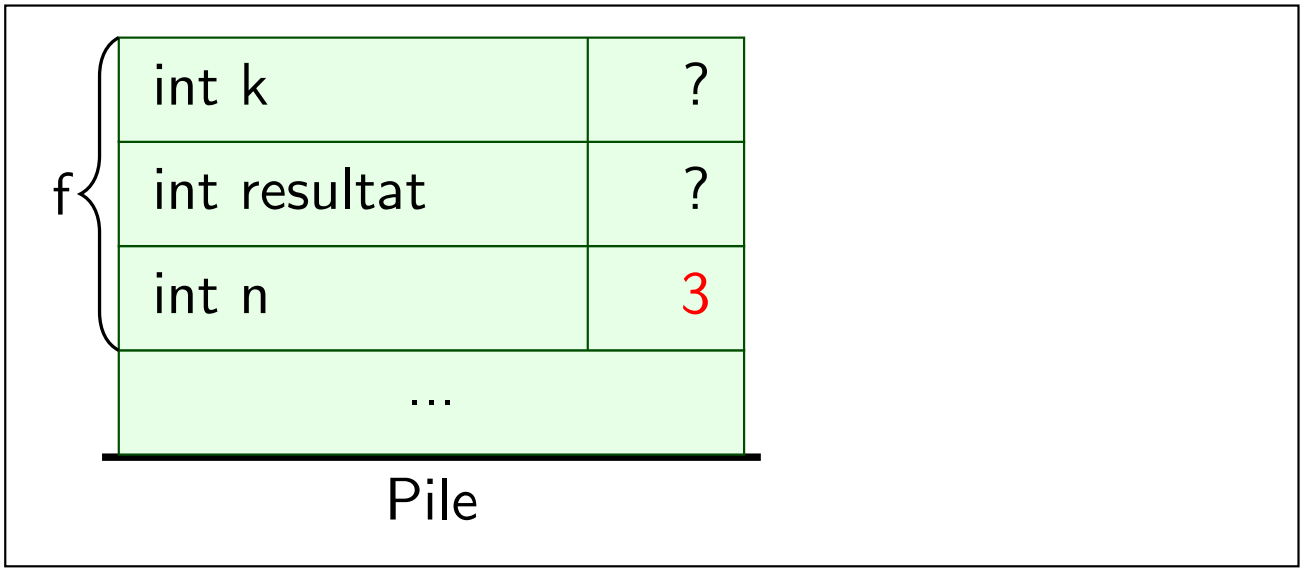
Évolution de la pile sur l'exemple :

3. Allocation de la mémoire sur la pile



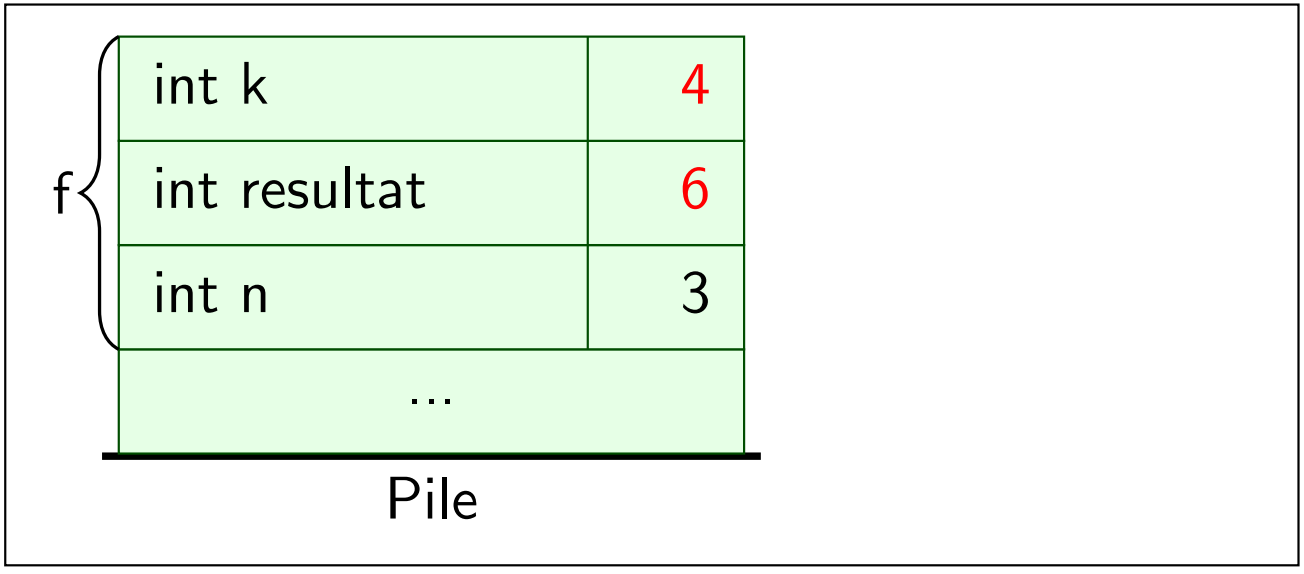
Évolution de la pile sur l'exemple :

4. Affectation du paramètre réel au paramètre formel



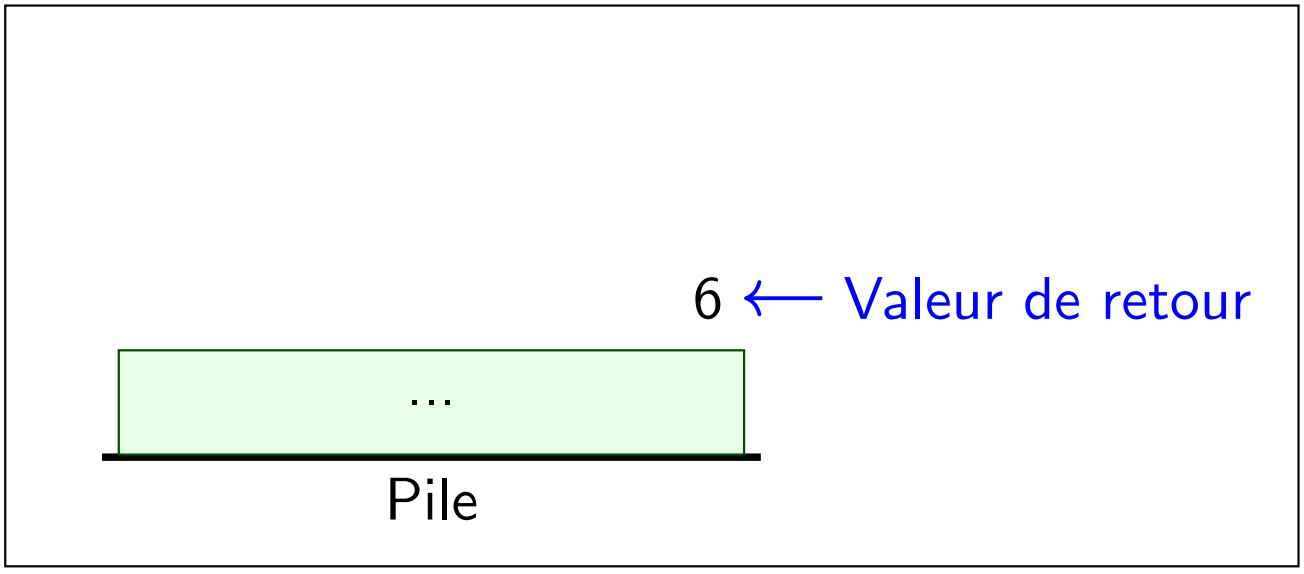
Évolution de la pile sur l'exemple :

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

Qu'affiche le (fragment de) programme suivant :

fonction-exemple-passage-par-valeur.cpp

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

int main() {
    int a, b;

    a = 1;
    b = incremente(a);

    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

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*

Portée des variables : un exemple

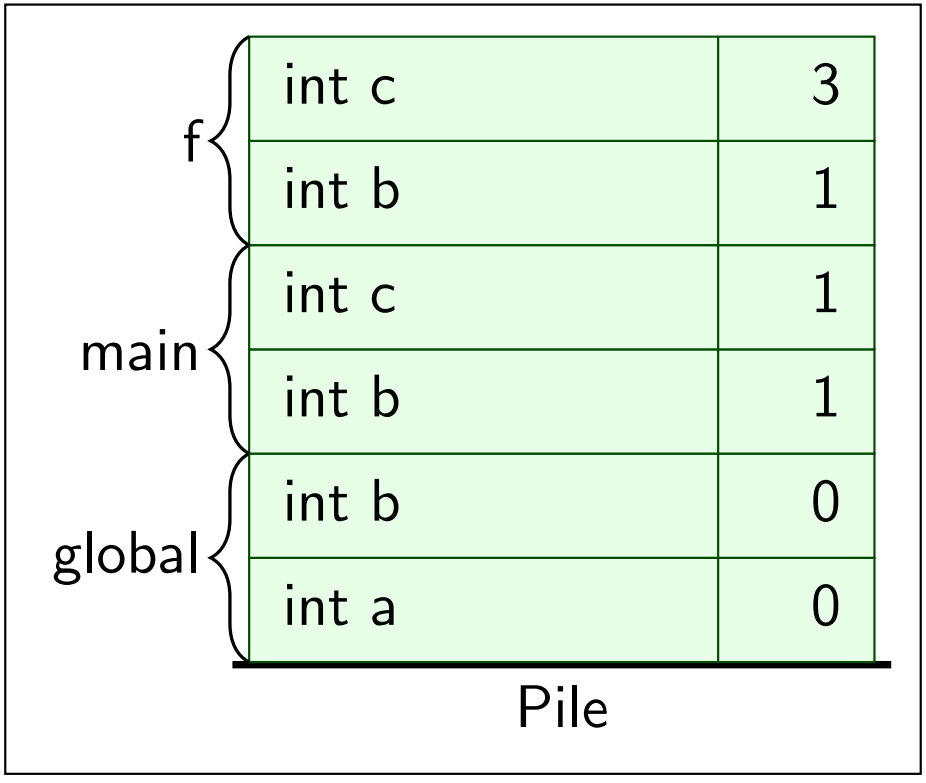
variables-locales-globales.cpp

```
int a = 0, b = 0;  // variables globales

int f(int b) {     // paramètre formel (donc local à f)
    int c = 3;     // variable locale à f
    return a + b + c;
}

int main() {
    int b = 1, c = 1; // variables locales à main
    a + b + c;     // b et c: locales à main, a: globale
    {
        long a = 2, c = 2;
        a + b + c; // a et c: locales au bloc, b: locale à main
    }
    a + b + c;     // b et c: locales à main, a: globale
    cout << f(b) << endl;
}
```

Portée des variables : un exemple (2)



Portée des variables

Contexte lexical

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc où elle est déclarée

Portée des variables

Contexte lexical

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc où elle est déclarée
- Elle peut masquer des variables issues des contextes englobants

Portée des variables

Contexte lexical

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc où elle est déclarée
- Elle peut masquer des variables issues des contextes englobants
- *Variable locale* : définie dans le bloc d'une fonction
- *Variable globale* : définie ailleurs (entête du programme)
- *Paramètre formel* : se comporte comme une variable locale

Portée des variables

Contexte lexical

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc où elle est déclarée
- Elle peut masquer des variables issues des contextes englobants
- *Variable locale* : définie dans le bloc d'une fonction
- *Variable globale* : définie ailleurs (entête du programme)
- *Paramètre formel* : se comporte comme une variable locale

Remarque

- Une variable locale à une fonction n'existe que le temps d'exécution de la fonction
- La valeur que cette variable peut avoir lors d'un appel à la fonction est perdue lors du retour au programme appelant et ne peut être récupérée lors d'un appel ultérieur

Variables globales

- Accessible à l'intérieur de toutes les fonctions

Variables globales

- Accessible à l'intérieur de toutes les fonctions

Attention !

- On peut modifier la valeur d'une variable globale
  Ceci est très fortement déconseillée (*effet de bord*)
- Une variable locale masque une variable globale du même nom
  Ceci est très fortement déconseillée (ambiguïté)
- On interdira ces pratiques dans le cadre de ce cours

F. Fonctions particulières

1. La fonction `main`
2. Procédures
3. Fonctions récursives

La fonction `main`

Exemple

bonjour.cpp

```
int main() {
    cout << "Bonjour !" << endl;
    return 0;
}
```

La fonction `main`

Exemple

bonjour.cpp

```
int main() {
    cout << "Bonjour !" << endl;
    return 0;
}
```

- Le programme principal est une fonction comme les autres !
- Ce programme renvoie une valeur entière
- Convention :
  - 0 si l'exécution du programme s'est déroulée normalement
  - Un entier différent de 0 en cas d'erreur
    Cet entier indique quel genre d'erreur s'est produite

La fonction `main` : paramètres ♣

bonjour-nom.cpp

```
int main(int argv, char ** args) {
    string nom1, nom2;
    nom1 = args[1];
    nom2 = args[2];

    cout << "Bonjour " << nom1 << " !" << endl;
    cout << "Bonjour " << nom2 << " !" << endl;
    return 0;
}
```

La fonction `main` : paramètres ♣

bonjour-nom.cpp

```
int main(int argv, char ** args) {
    string nom1, nom2;
    nom1 = args[1];
    nom2 = args[2];

    cout << "Bonjour " << nom1 << " !" << endl;
    cout << "Bonjour " << nom2 << " !" << endl;
    return 0;
}
```

À l'exécution :

```
> bonjour-nom Jean Paul
Bonjour Jean !
Bonjour Paul !
```

La fonction `main` : paramètres ♣

bonjour-nom.cpp

```
int main(int argv, char ** args) {
    string nom1, nom2;
    nom1 = args[1];
    nom2 = args[2];

    cout << "Bonjour " << nom1 << " !" << endl;
    cout << "Bonjour " << nom2 << " !" << endl;
    return 0;
}
```

À l'exécution :

```
> bonjour-nom Jean Paul
Bonjour Jean !
Bonjour Paul !
```

Note

Le `char **` est un résidu des chaînes de caractères en C.

Vous verrez au second semestre ce que font les trois premières lignes.

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*

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

```
void affiche_rectangle(int n) {
    for ( int k = 0; k < n; k++ ) {
        cout << "**********" << endl;
    }
}
```

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

```
void affiche_rectangle(int n) {
    for ( int k = 0; k < n; k++ ) {
        cout << "**********" << endl;
    }
}
```

Remarques

- Cette fonction ne renvoie rien
- On le dénote en C++ par le type `void`
- Dans d'autres langages on distingue *fonctions* et *procédures*

Fonctions récursives ♣

Exercice

Exécuter pas-à-pas l'exécution du programme suivant pour $n = 4$:

fonction-factorielle-recursive.cpp

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

int main() {
    int n;
    cin >> n;
    cout << n << "! = " << factorielle(n) << endl;
    return 0;
}
```

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