Introduction à la Programmation

Benoit Donnet

Année Académique 2022 - 2023

Agenda

Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
  - Macro

Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
  - Macro

Principe

- Un programme typique comporte plusieurs dizaines de milliers de lignes de code
  - impossible de tout mettre dans `int main(){}`
    - ✓ illisible
    - ✓ trop de variables
- Comment faire pour gérer son code de manière optimale?

Principe (2)

- Un programme peut être découpé en **modules**
  - morceau de code qui est écrit indépendamment du programme principal et peut être *invoqué* (ou *appelé*) à partir de plusieurs endroits du programme
- Un module peut
  - retourner un résultat
    - ✓ **fonction**
    - ✓ exemples
      - `fopen()`
      - `fscanf()`
  - ne pas retourner de résultat
    - ✓ **procédure**
    - ✓ exemple
      - `printf()`

Principe (3)

- Avantages d'une découpe en modules?
  - *approche systémique*
    - ✓ chaque module se concentre sur un sous-problème particulier, indépendamment du reste du programme
  - *lisibilité*
    - ✓ il est plus facile de lire/comprendre un module d'une dizaine de lignes qu'un programme unique de 10.000 lignes
  - *réutilisabilité*
    - ✓ un même module peut être réutilisé plusieurs fois dans un programme
    - ✓ un même module peut être réutilisé plusieurs fois dans des programmes différents
      - notion de **bibliothèque**
      - cfr. Compilation Séparée (Slides 32 → 47)
      - cfr. INFO0030

Principe (4)

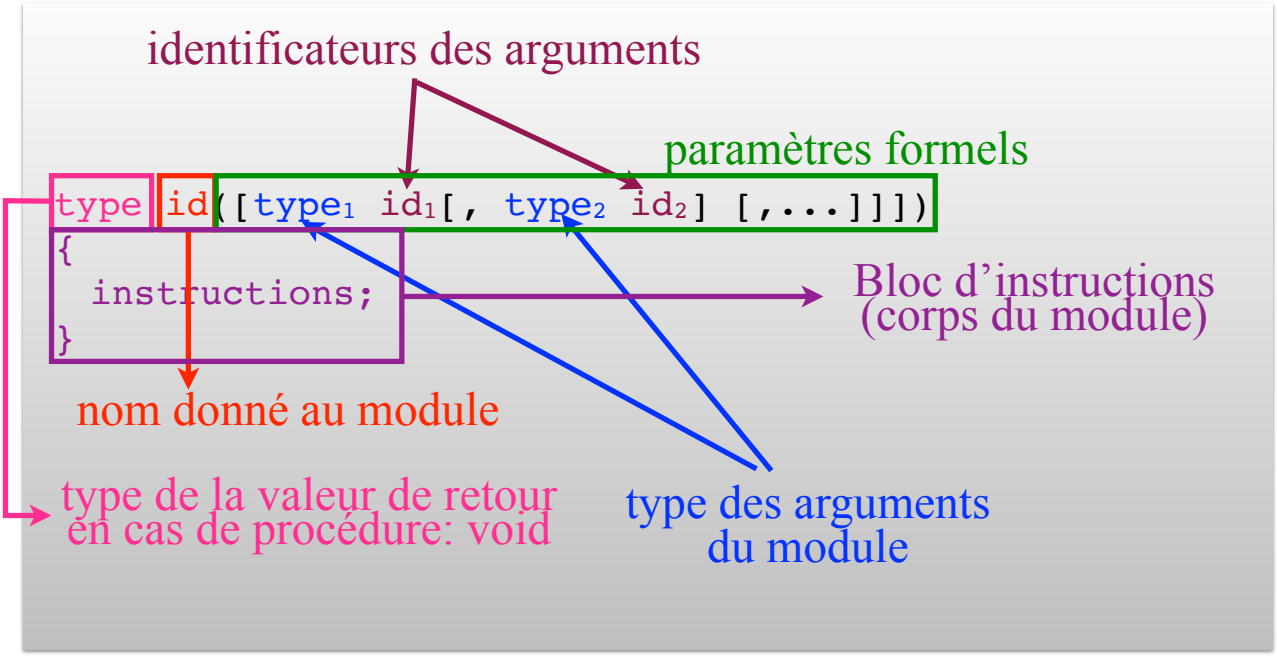
- Fonctionnement?
  - le code invoque un module
    - ✓ celui qui invoque est appelé **code appelant**
  - pendant l'exécution du module, l'exécution du code appelant est suspendue
    - ✓ c'est le module qui a la main
  - le code appelant reprend son exécution lorsque le module invoqué est terminé
- Un module peut disposer d'**arguments**
  - données en entrée utilisées par le module
    - ✓ **paramètres formels**
    - ✓ ils n'ont d'existence que dans le module où ils sont définis
  - ces données sont passées au module lors de l'invocation
    - ✓ **paramètres effectifs**
  - cfr. Chap. 7 pour le détail sur le *passage de paramètres*

Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
    - ✓ Déclaration
    - ✓ Portée des Variables
    - ✓ Invocation
    - ✓ Application
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
  - Macro

Déclaration

- Une fonction/procédure est déclarée comme suit:



Déclaration (2)

- **Prototype** d'une fonction/procédure
  - type de retour
  - identificateur
  - liste des paramètres formels
- On parle aussi de **signature**
- A l'intérieur du bloc d'instructions, on procède "comme d'habitude"
  - déclaration de variables
    - ✓ **variables locales**
    - ✓ cfr. Portée des Variables (Slide 18)
  - instructions

Déclaration (3)

- Exemple 1
  - procédure qui affiche à l'écran le contenu d'un tableau d'entiers
  - cfr. Chap. 5 pour la construction du code

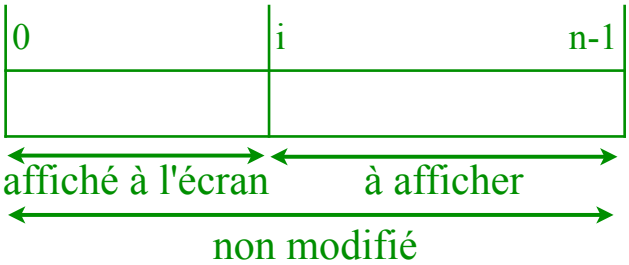
rien à retourner

identificateur

paramètres formels

```
void afficher_tableau(int tab[], int n){
  int i;
  printf("[ ");          //Inv: tab:
  for(i=0; i<n; i++)
    printf("%d ", tab[i]);
  printf("]\n");
}//fin affiche_tableau()
```

variable locale



Déclaration (4)

- Exemple 2
  - fonction qui retourne x^7

retourne une valeur de type `int`

identificateur

```
int puissance7(int x){
  int resultat;

  resultat = x*x*x*x*x*x*x;

  return resultat;
}//fin puissance7()
```

paramètre formel

renvoi du résultat au code appelant

Déclaration (5)

- Dans le corps d'un module, l'instruction
  - **`return`** `[expression];`
  - termine immédiatement l'exécution du module
- Si fonction
  - l'instruction **`return`** est obligatoire et est suivie d'une expression dont le type correspond au type de retour de la fonction
  - l'expression est évaluée et la valeur évaluée est retournée au code appelant
- Si procédure
  - l'instruction **`return`** est facultative
  - si présente, elle ne peut pas avoir d'expression

Déclaration (6)

- Où placer la définition d'un module?
- En C, il est interdit de définir un module dans le corps d'un autre module
  - un programme se compose donc d'une suite de définition de modules
  - les modules sont définis entre les dérives de compilation et `main()`

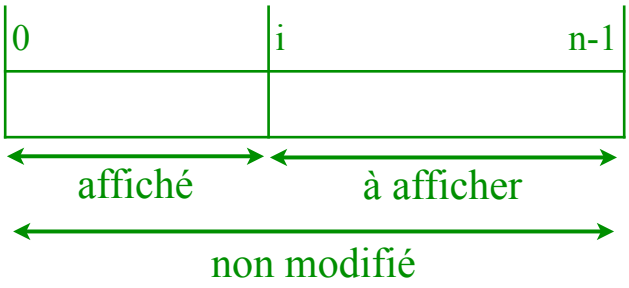
Déclaration (7)

- La définition `int main() {...}` est celle de la fonction "main"
  - point d'entrée d'un programme C
  - c'est la fonction qui est invoquée dès le début de l'exécution du programme
- Pourquoi retourne-t-elle une valeur entière?
  - code de diagnostic renvoyé en fin d'exécution
  - le code "0" correspond à une exécution sans erreur
  - dorénavant, nous allons terminer les "main" par l'instruction `return 0;`

Déclaration (8)

- Exemple

```
#include <stdio.h>

void afficher_tableau(int tab[], int n){
  int i;

  printf("[ ");
  for(i=0; i<n; i++)
    printf("%d ", tab[i]);
  printf("]\n");
}//fin affiche_tableau()

int main(){
  //code du main

  return 0;
}//fin main()
```



Portée des Variables

- Quid des variables déclarées (et utilisées) dans le corps d'un module?
  - elles n'ont aucune existence en dehors du module
- La **portée d'une variable** détermine le bloc d'instructions dans lequel la variable est utilisable
  - par défaut, la portée est toujours limitée au bloc dans lequel on définit la variable
  - on peut déclarer une variable dans n'importe quel bloc
- Les paramètres formels d'un module peuvent être vus comme des variables
  - dont la portée est limitée au corps du module
  - les valeurs sont initialisées à l'aide des paramètres fournis lors de l'appel du module

Invocation

- Comment invoquer le module déclaré?
- Si un module `m` a été correctement déclaré, alors il peut être invoqué comme suit

```
m(expr1, expr2, ...);    paramètres effectifs
identificateur du module
```

- Les paramètres effectifs sont des expressions dont le type correspond à celui des paramètres formels de `m`
- Lors de l'invocation, on **ne doit pas** indiquer
  - le type de retour (`void` ou autre)
  - le type des paramètres

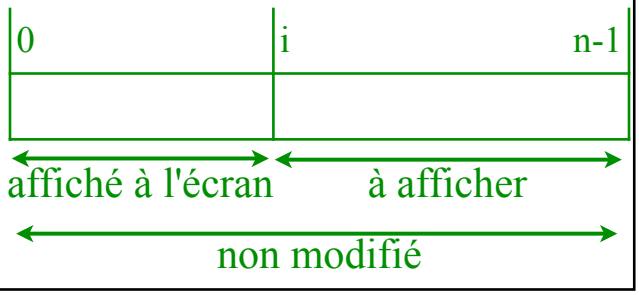
Invocation (2)

- Si `m` est une fonction
  - `m` est une valeur à droite
    - ✓ peut se trouver à droite d'une affectation
  - `m` est vu comme une expression
    - ✓ dont l'évaluation est égale à la valeur retournée par `m` à la fin de son exécution
  - exemple
    - ✓ utilisation de `fopen()`
    - ✓ utilisation de `fscanf()`
- Si `m` est une procédure
  - `m` ne peut pas se trouver à droite d'une affectation
  - `m` est vu comme une expression n'ayant pas de valeur
  - exemple
    - ✓ utilisation de `printf()`

Invocation (3)

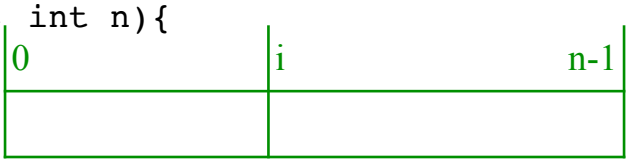
- Exemple

```
#include <stdio.h>

int puissance7(int x){
  int resultat = x*x*x*x*x*x*x;

  return resultat;
}//fin puissance7()

void afficher_tableau(int tab[], int n){
  int i;

  printf("[ ");        //Inv: tab:
  for(i=0; i<n; i++)
    printf("%d ", tab[i]);
  printf("]\n");
}//fin affiche_tableau()
```

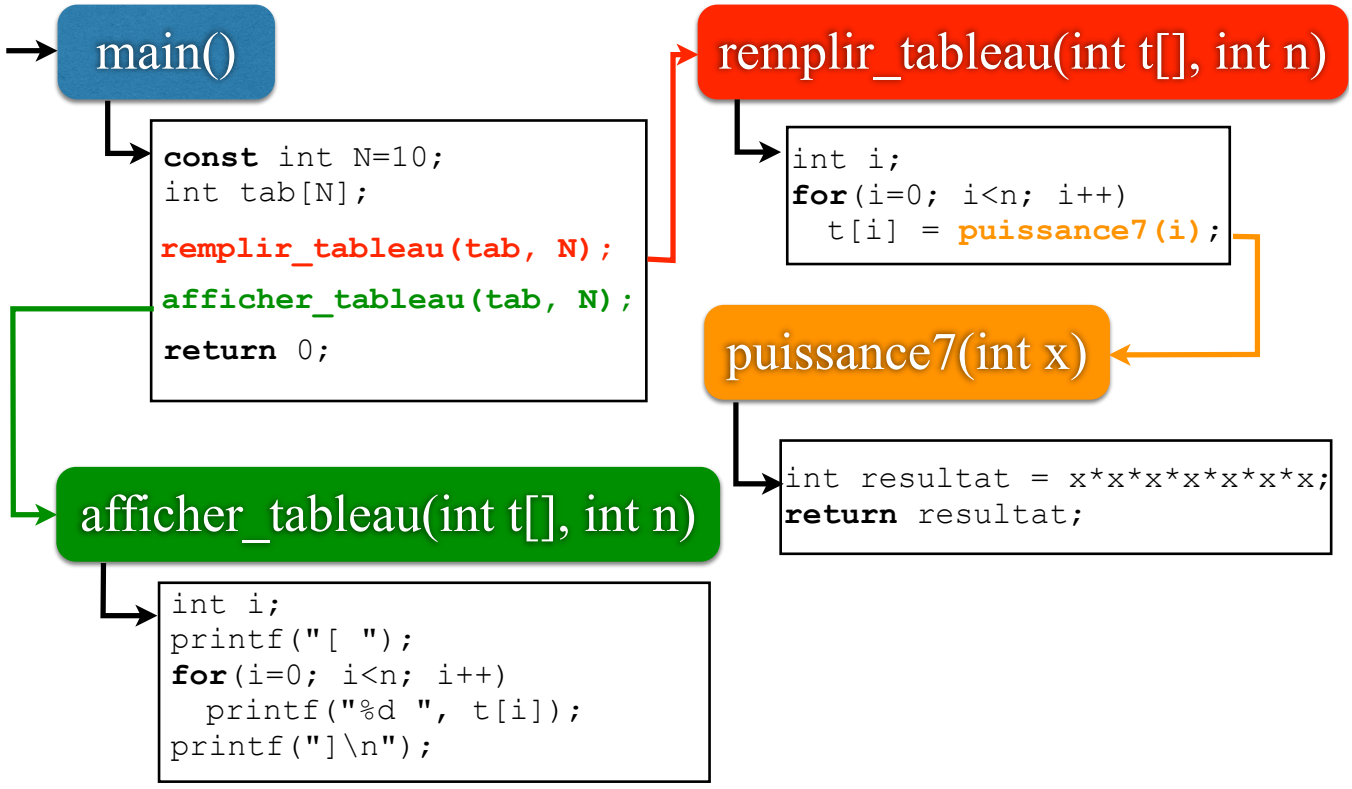


Invocation (4)

```
#include <stdio.h>

void remplir_tableau(int t[], int n){
  int i;

  for(i=0; i<n; i++)        //Inv: t:
    t[i] = puissance7(i);
}//fin remplir_tableau()

int main(){
  const int N = 10;
  int tab[N];

  remplir_tableau(tab, N);
  afficher_tableau(tab, N);

  return 0;
}//fin programme
```



Invocation (5)



Application

- Application:
  - approximation de sinus(x) par un développement en série de Taylor

$$sin(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}$$

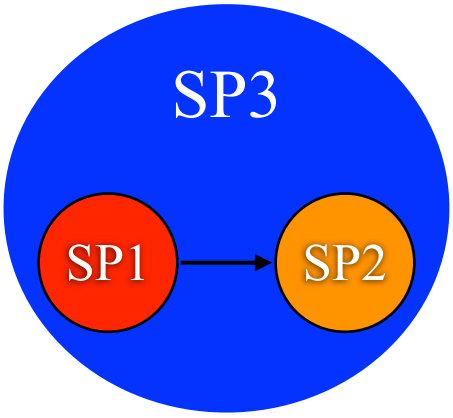
Application (2)

- Définition du problème
  - Input
    - ✓ `x`
      - valeur pour laquelle on cherche à approximer le sinus
    - ✓ `precision`
      - nombre de termes dans la somme
  - Output
    - ✓ approximation de sinus(x)
  - Objets Utilisés
    - ✓ `x` est un angle, exprimé en radians et $\in [0; 2\pi]$
      - `double x;`
    - ✓ `precision` est une valeur entière ≥ 0
      - `unsigned int precision;`

Application (3)

- Analyse du problème
  - SP1: calcul de la factorielle
  - SP2: calcul de a^b
  - SP3: développement en série de Taylor
- Enchaînement des SPs
  - (SP1 → SP2) ⊂ SP3

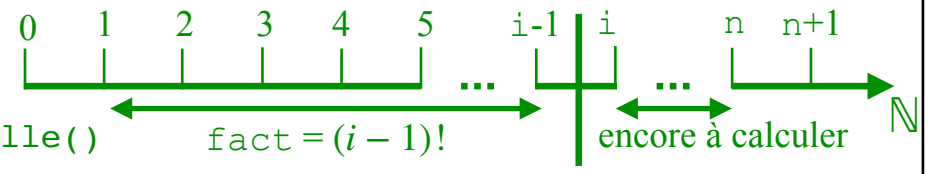
$$\sum_{n=0}^{precision-1} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$



Application (4)

- Sous-Problème 1: calcul de factorielle
  - définition du SP
    - ✓ Input
      - `n`, le nombre pour lequel il faut calculer la factorielle
    - ✓ Output
      - `n!`
    - ✓ Objet Utilisé
      - `int n`

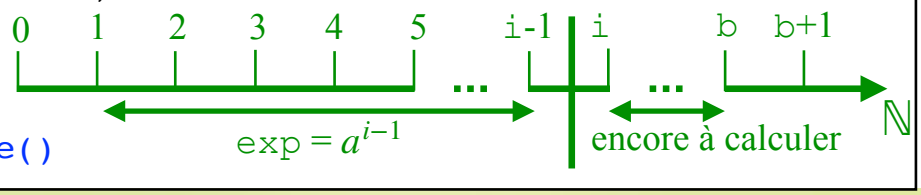
```
int factorielle(int n){
  int fact = 1, i;

  for(i=1; i<=n; i++)
    fact *= i;

  return fact;
}//fin factorielle()
```



Application (5)

- Sous-Problème 2: calcul de a^b
  - définition du SP
    - ✓ Input
      - a, la base
      - b, l'exposant
    - ✓ Output
      - a^b
    - ✓ Objets Utilisés
      - `double a`
      - `int b`

```
double puissance(double a, int b){
  double exp = 1.0;
  int i;

  for(i=1; i<=b; i++)
    exp *= a;

  return exp;
}//fin puissance()
```



Application (6)

- Sous-Problème 3: développement série de Taylor
  - construction de l'Invariant

$$\sum_{i=0}^{precision-1} \frac{(-1)^i}{(2i+1)!}x^{2i+1} =$$

$$\underbrace{\frac{(-1)^0}{(1)!}x^1 + \ldots + \frac{(-1)^{n-1}}{(2(n-1)+1)!}x^{2(n-1)+1}}_{\texttt{sin}\text{ contient l'approximation du sinus avec }\texttt{n}\text{ termes}} + \underbrace{\frac{(-1)^n}{(2n+1)!}x^{2n+1} + \ldots + \frac{(-1)^{p-1}}{(2(p-1)+1)!}x^{2(p-1)+1}}_{\text{encore à sommer}}$$

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Fonction de Terminaison
  - precision - n

Application (7)

- Sous-Problème 3: développement série de Taylor

```
double sinus_taylor(double x, unsigned int precision){
  double sin = 0;
  int n;

  for(n=0; n<precision; n++){
    double num = puissance(-1, n);
    int den = factorielle(2*n+1);
    double tmp = puissance(x, 2*n+1);

    sin += (num/den) * tmp;
  }//fin for - n
                //Inv: sin = Σ_{i=0}^{n-1} (-1)^i/(2i+1)! x^{2i+1}, 0 ≤ n ≤ precision
  return sin;
}//fin sinus_taylor()
```

Application (7)

- Programme

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

//déclaration des fonctions (cfr. slides 27 → 29)

int main(){
  double sinus = sinus_taylor(0.9, 8);

  printf("sinus(0.9): %f %f\n", sinus, sin(0.9));

  return 0;
}//fin programme
```


Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
    - ✓ Principe
    - ✓ Application
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
  - Macro

Principe

- Pour pouvoir invoquer un module, il est nécessaire que celui-ci soit connu
- Le compilateur peut prendre connaissance du module
  - *implicitement* en le déclarant entièrement
    - ✓ cfr. Slide 10 du Chapitre 6
  - *explicitement* en fournissant uniquement une déclaration du prototype
- Format d'une déclaration explicite

```
type id ([type1 id1[, type2 id2] [,...]]);

type id (void);
```


Principe (2)

- La surcharge de module est interdite en C
  - deux modules doivent avoir des identificateurs différents
- L'ensemble des prototypes est regroupé au sein d'un fichier particulier
  - **header**
  - `source.h`
- Un tel fichier peut être incorporé à un fichier source classique grâce à une directive de pré-traitement
  - *dérive de compilation* ou *preprocessing directive*
  - 2 formes
    - ✓ `#include <source.h>` ⇒ header standard
    - ✓ `#include "source.h"` ⇒ header fourni par le programmeur

Principe (3)

- Un des avantages des headers est de pouvoir définir des modules qui pourront être réutilisées par la suite
  - dans le programme
  - dans d'autres programmes
- Exemple
  - `stdio.h`
    - ✓ contient les prototypes des modules permettant de gérer les entrées/sorties (standard input and output), dont `printf()` et `scanf()`
    - ✓ `#include <stdio.h>` permet au compilateur de connaître les modalités d'invocation de ces modules
- On peut définir autant de headers qu'on veut

Principe (4)

- Si le header ne contient que les prototypes, il faut pouvoir associer, à ces prototypes, le corps des modules
- A tout header est associé un autre fichier qui, lui, contient le corps des modules
  - fichier d'implémentation (**module**)
  - `source.c`
- Le header et le module ont (généralement) le même nom
  - le fichier d'implémentation inclut le header associé
  - et tous les headers nécessaires

Application

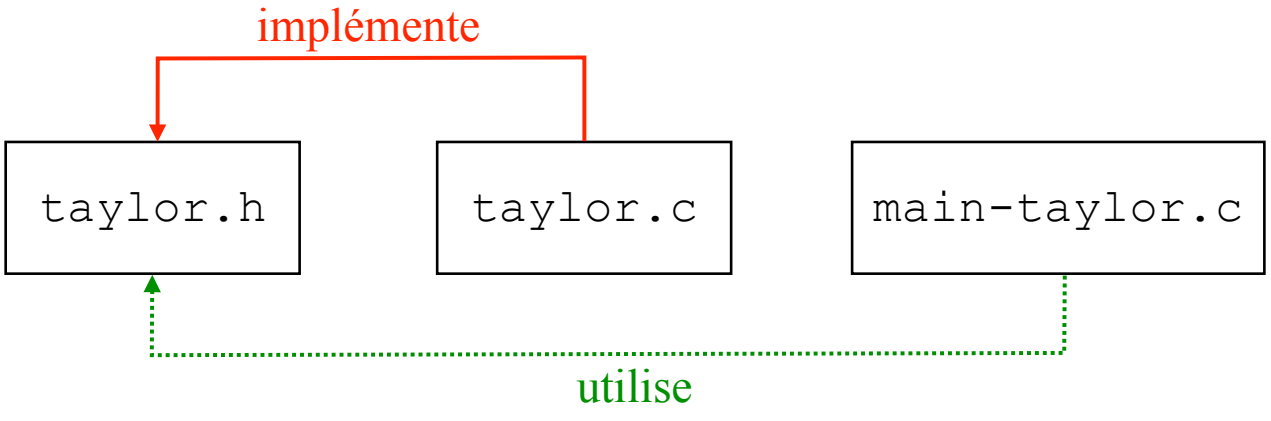
- Application:
  - approximation de *sinus(x)* et *cosinus(x)* par un développement en série de Taylor

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

Application (2)

- Architecture du code



Application (3)

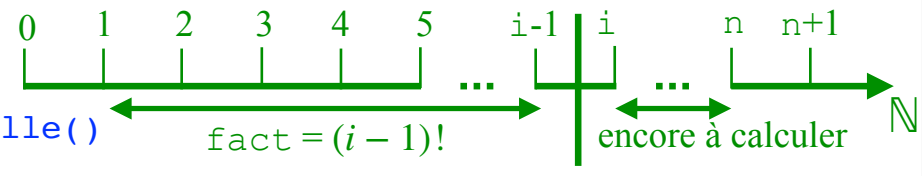
- Header
  - taylor.h

```c
// Calcule la factorielle de n (n!)
int factorielle(int n);

// Calcule a^b
double puissance(double a, int b);

//Calcule sin(x), via une approximation en série de Taylor
double sinus_taylor(double x, unsigned int precision);

//Calcule cos(x), via une approximation en série de Taylor
double cosinus_taylor(double x, unsigned int precision);
```


Application (4)

- Module
  - taylor.c

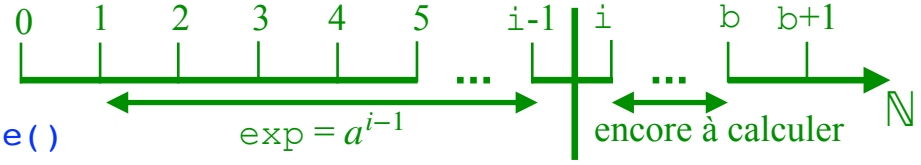
```c
#include "taylor.h"

int factorielle(int n){
  int fact = 1, i;

  for(i=1; i<=n; i++)
      fact *= i;

  return fact;
}//fin factorielle()
```



Application (5)

- Module (cont.)
  - taylor.c

```
double puissance(double a, int b){
  double exp = 1.0;
  int i;

  for(i=1; i<=b; i++)
    exp *= a;

  return exp;
}//fin puissance()
```



Application (6)

- Module (cont.)
  - taylor.c

```
double sinus_taylor(double x, unsigned int precision){
  double sin = 0;
  int n;

  for(n=0; n<precision; n++){
    double num = puissance(-1, n);
    int den = factorielle(2*n+1);
    double tmp = puissance(x, 2*n+1);

    sin += (num/den) * tmp;
  }//fin for - n
                //Inv: sin = $\sum_{i=0}^{n-1} \frac{(-1)^i}{(2i+1)!} x^{2i+1}, 0 \leq n \leq precision$
  return sin;
}//fin sinus_taylor()
```

Application (7)

- Module (cont)
  - taylor.c

```
double cosinus_taylor(double x, unsigned int precision){
  double cos = 0;
  int n;

  for(n=0; n<precision; n++){
    double num = puissance(-1, n);
    int den = factorielle(2*n);
    double tmp = puissance(x, 2*n);

    cos += (num/den) * tmp;
  }//fin for - n
                    //Inv: cos = \sum_{i=0}^{n-1} \frac{(-1)^i}{(2i)!} x^{2i}, 0 \leq n \leq precision
  return cos;
}//fin cosinus_taylor()
```


Application (8)

- Programme principal
  - main-taylor.c

```
#include <stdio.h>
#include <math.h>
#include "taylor.h"

int main(){
  double sinus = sinus_taylor(0.9, 8);
  double cosinus = cosinus_taylor(0.9, 8);

  printf("sinus(0.9): %f %f\n", sinus, sin(0.9));
  printf("cosinus(0.9): %f %f\n", cosinus, cos(0.9));

  return 0;
}//fin programme
```

Application (9)

- Comment compiler?
  - Tentative 1

```
$> gcc -o main main-taylor.c


Undefined symbols for architecture x86_64:
  "_sinus_taylor", referenced from:
      _main in ccM8LfpM.o
  "_cosinus_taylor", referenced from:
      _main in ccM8LfpM.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
```


Application (10)

- Comment compiler?
  - Tentative 2

```
$> gcc -o main main-taylor.c taylor.c

$>
```

```
$> gcc -o main *.c

$>
```

Exercices

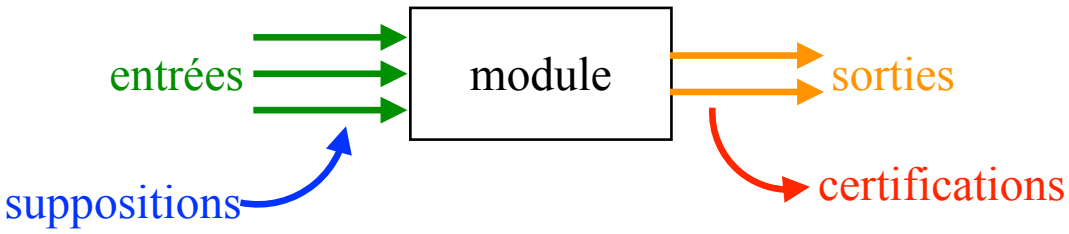
- Ecrire
  - une fonction f1 qui se contente d'afficher "bonjour"
  - une fonction f2 qui affiche "bonjour" un nombre de fois égal à la valeur reçue en argument
  - une fonction f3 qui fait la même chose que f2 mais renvoie une valeur entière (0) en retour.
  - un programme qui appelle chacune des fonctions après les avoir déclarées et implémentées dans un module (fonction.h et fonction.c)
- Pour un ménage X avec un revenu total R et n membres du foyer, l'impôt est de
  - 10% de R si R/n < 500€
  - 20% de R sinon
  - Ecrire une fonction impot qui calcule l'impôt en fonction de R et n
  - Ecrire une fonction revenu_net qui calcule le revenu net d'un ménage après paiement de l'impôt en fonction de R et n.
  - Ecrire un programme qui saisit R et n au clavier et affiche l'impôt et le revenu net

Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
    - ✓ Principe
    - ✓ Spécifications
    - ✓ Programmation Défensive
    - ✓ Application
  - Variable Statique
  - Variable Globale
  - Macro

Principe

- La programmation par contrat est une méthodologie de développement à appliquer à un module
  - avant d'écrire l'implémentation d'un module
  - revient à "formaliser" la définition d'un (sous-)problème
- On définit:
  - les **suppositions** sur les arguments en entrée
  - les **certifications** après exécution du module

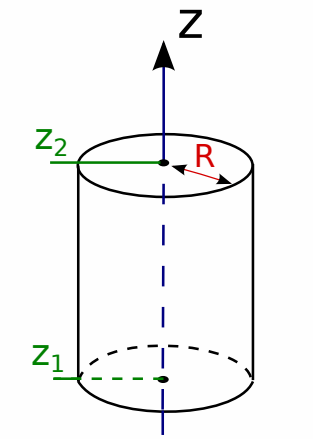


Principe (2)

- Avantage(s)
  - on empêche les cas particuliers/effets de bord
    - ✓ à tout le moins, on ne les oublie pas!
  - on auto-documente le module
    - ✓ cfr. INFO0030, Partie 2, Chap. 4
  - c'est une aide aux tests, garantie de la validité
    - ✓ réduction des bugs
    - ✓ cfr. INFO0030, Partie 2, Chap. 3
- Inconvénient(s)
  - travail supplémentaire en amont

Principe (3)

- Exemple
  - calcul du volume d'un cylindre de rayon R entre $z = [z_1, z_2]$
  - $\pi \times R^2 \times (z_2 - z_1)$



Principe (4)

- Quelles sont les suppositions sur les arguments?
  - z_1 et z_2 représentent des coordonnées telles que $z_2 > z_1$
  - R représente le rayon tel que $R > 0$
- Quelles sont les certifications après exécution du module?
  - renvoi un volume (> 0) correspondant au cylindre décrit

Principe (5)

- Code

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

float volume(float z1, float z2, float R){
  return M_PI * R * R * (z2 - z1):
}//fin volume()

int main(){
  float v1 = volume(1, 2, 0.5);
  float v2 = volume(0, 10, 8);
  float v3 = volume(-4, 8, 1);
  float v4 = volume(5, 1, 1);
  float v5 = volume(1, 4 ,-1);

  return 0;
}//fin programme
```


Spécifications

- Dans un programme, un module possède
  - une **interface** qui regroupe
    - ✓ son nom
    - ✓ le nombre, le type, et la signification de ses paramètres
    - ✓ le type et la signification de son éventuelle valeur de retour
    - ✓ une description du travail qu'il effectue
    - ✓ des contraintes d'utilisation éventuelles
  - une **implémentation**
    - ✓ suite d'instructions qui forment le corps du module
- Il est donc possible d'utiliser un module sans en connaître son implémentation

Spécifications (2)

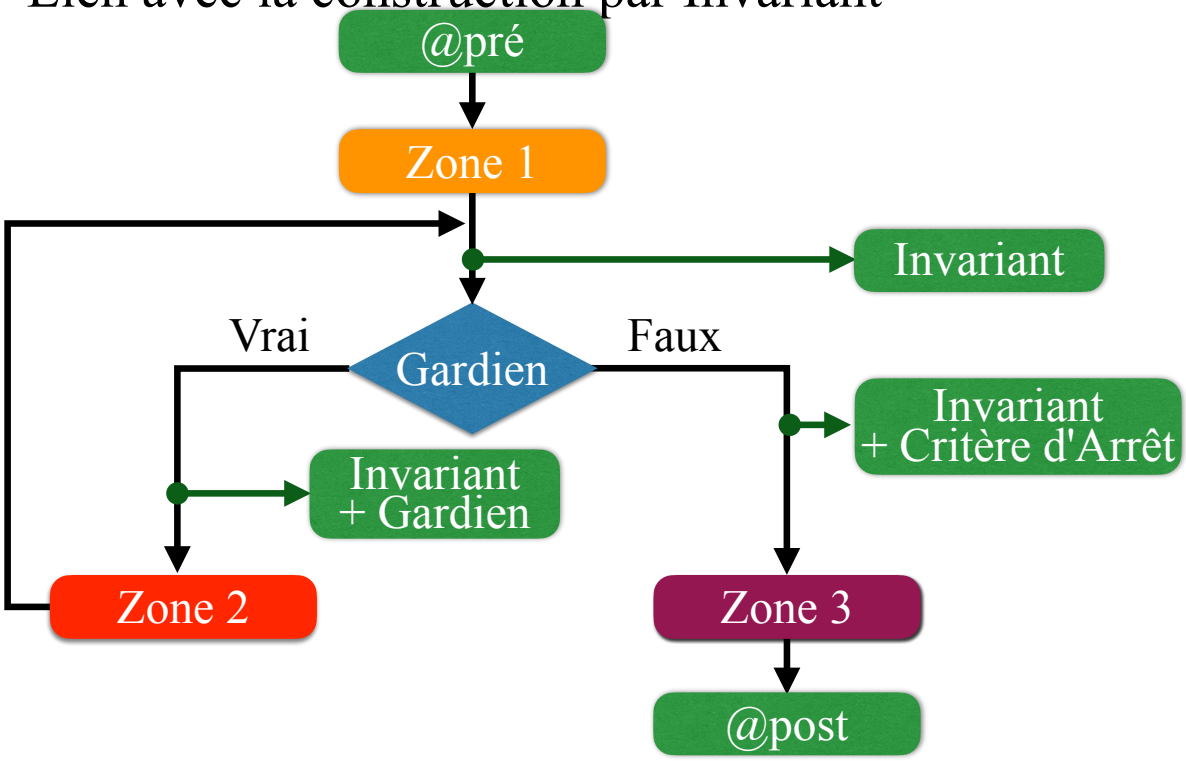
- La description du travail et les contraintes d'utilisation sont expliquées grâce à des **spécifications**
- Contrat logiciel qui lie
  - ✓ le programmeur du module
  - ✓ l'utilisateur du module
- Une spécification est définie en deux temps
  - **précondition**
    - ✓ caractérise les conditions initiales d'exécution du module
    - ✓ en particulier les données (paramètres)
    - ✓ doit être satisfaite avant l'appel au module
    - ✓ *supposition*
  - **postcondition**
    - ✓ caractérise les conditions finales d'exécution du module
    - ✓ en particulier le résultat
    - ✓ sera satisfaite après l'appel
    - ✓ *certification*

Spécifications (3)

- Qualités d'une spécification
  - simple, claire, précise
  - complète, non ambiguë
  - non-contradictoire

Spécifications (4)

- Lien avec la construction par Invariant

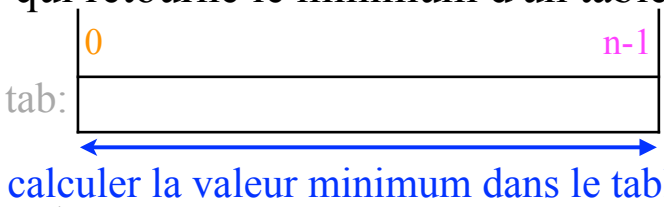


Spécifications (5)

- Comment écrire correctement un module et sa spécification?
- **Faire un dessin** (quand applicable) et répondre à 3 questions
  1. quels sont les paramètres (nom, type, signification) nécessaires pour atteindre l'objectif du module?
     - ✓ prototype du module
  2. quel est l'objectif de mon module?
     - ✓ postcondition
  3. quelles sont les contraintes sur ces paramètres?
     - ✓ précondition

Spécifications (6)

- Exemple 1
  - module qui retourne le minimum d'un tableau d'entiers



calculer la valeur minimum dans le tableau

- Trois questions
  1. quels sont les paramètres?
     - ✓ un tableau `tab` de `n` valeurs entières
  2. quel est l'objectif de la fonction?
     - ✓ obtenir le minimum de `tab`
     - ✓ postcondition: retourne le minimum du tableau `tab`
  3. quelles sont les contraintes sur les paramètres?
     - ✓ `n` ne peut être négatif ou nul
     - ✓ `tab` doit exister et contenir des valeurs
     - ✓ précondition: `tab` initialisé, `n` > 0

Spécifications (7)

- Exemple 1
  - fonction qui retourne le minimum d'un tableau d'entier

```
/*
 * @pre: tab est initialisé, n > 0
 * @post: minimum vaut le minimum du tableau tab
 */
int minimum(int tab[], int n){
  int min = tab[0], i;

  //Inv: min = MINIMUM{tab[0], ..., tab[i-1]} ∧ 1 ≤ i ≤ n
  for(i=1; i<n; i++){
    if(tab[i]<min)
      min = tab[i];
  }//fin for - i
  //Inv: min = MINIMIM{tab[0], ..., tab[i-1]} ∧ 1 ≤ i ≤ n
  //      ∧ i≥n
  //⇒ min=MINIMUM{tab[0], ..., tab[n-1]}
  return min;
}//fin minimum()
```

Spécifications (8)

- Exemple 2
  - fonction qui calcule le volume d'un cylindre de rayon R entre $z = [z_1, z_2]$
- Trois questions
  1. quels sont les paramètres?
     - ✓ les coordonnées `z1`, `z2`
     - ✓ le rayon `R`
  2. quel est l'objectif de la fonction?
     - ✓ retourner le volume du cylindre décrit
     - ✓ postcondition: le volume du cylindre (>0) de rayon `R` et de coordonnées (`z1`, `z2`)
  3. quelles sont les contraintes sur les paramètres?
     - ✓ `R` ne peut être que strictement positif
     - ✓ `z2` doit être plus supérieur à `z1`
     - ✓ précondition: `R` > 0, `z2` > `z1`

Spécifications (9)

- Exemple 2
  - fonction qui calcule le volume d'un cylindre de rayon R entre $z = [z_1, z_2]$

```
/*
 * @pre: R>0, z2 > z1
 * @post: volume vaut le volume du cylindre (>0) de rayon
 *        R et de coordonnées (z1, z2)
 */
float volume(float z1, float z2, float R){
  return M_PI * R * R * (z2 - z1):
}//fin volume()
```

Spécifications (10)

- Exemple 3
  - approximation de sinus(x) par un développement en série de Taylor
- On dispose de 3 sous-problèmes
  - calcul de la factorielle
  - calcul de la puissance
  - calcul du sinus

Spécifications (11)

```
/*
 * @pre: n ≥ 0
 * @post: factorielle vaut n!
 */
int factorielle(int n);

/*
 * @pre: b ≥ 0
 * @post: puissance vaut a^b
 */
double puissance(double a, int b);

/*
 * @pre: ?????
 * @post: ?????
 */
double sinus_taylor(double x, int precision);
```

Progra. Défensive

- Une spécification est un contrat logiciel entre
  - le programmeur du module
  - l'utilisateur du module
- Au fond, qu'est-ce qui nous garantit que l'utilisateur va bien lire les spécifications?
- Quid si le programme est exécuté avec des données ne respectant pas les spécifications?
  - quid si `factorielle(-5)`?
  - on ne peut rien dire du résultat!
- Solution
  - *programmation défensive*

Progra. Défensive (2)

- Programmation défensive
  - vérifier certaines préconditions dans le corps du module même
  - il s'agit donc d'une programmation "prudente"
- Que faire en cas de non-respect d'une précondition?
  - message d'erreur à l'écran et arrêt du programme
  - résultat "spécial"
    - ✓ cfr. le **`return`** dans la fonction `main()`
  - utiliser `void assert(int expression)`
    - ✓ c'est ce qu'on va utiliser dans le cadre du cours

Progra. Défensive (3)

- La procédure `void assert(int)` est définie dans `assert.h`
  - dérive de compilation: `#include <assert.h>`
  - permet l'inscription, à l'écran, d'un diagnostic de fonctionnement
- Fonctionnement?
  - si l'expression évaluée par `assert(int)` est vraie
    - ✓ le programme poursuit normalement son exécution
  - sinon
    - ✓ un message d'erreur est envoyé à l'écran
    - ✓ l'exécution du programme est terminée automatiquement

Progra. Défensive (4)

- Exemple 1
  - calcul de factorielle

```
#include <assert.h>
```
dérive de compilation

```
/*
 * @pre: n ≥ 0
 * @post: factorielle vaut n!
 */
int factorielle(int n){
  assert(n>=0);
  int fact = 1, i;
  for(i=1; i<=n; i++)
      fact *= i;

  return fact;
}//fin factorielle()
```
vérifie la précondition sur `n`

Progra. Défensive (5)

```
int main(){
  printf("%d\n", factorielle(5));
  printf("%d\n", factorielle(0));
  printf("%d\n", factorielle(-10));

  return 0;
}//fin programme
```

```
$> gcc -o factorielle factorielle.c
$>./factorielle
120
1
Assertion failed: (n>=0), function factorielle, file
factorielle.c, line 9.
Abort trap: 6
```


Progra. Défensive (6)

- Exemple 2
  - fonction qui calcule le volume d'un cylindre de rayon R entre $z = [z_1, z_2]$

```
#include <assert.h>

/*
 * @pre: R>0, z2 > z1
 * @post: volume vaut le volume du cylindre (>0) de rayon
 *        R et de coordonnées (z1, z2)
 */
float volume(float z1, float z2, float R){
  assert((z2>z1) && (R>0));

  return M_PI * R * R * (z2 - z1):
}//fin volume()
```

Application

- Tri d'un tableau
- Les algorithmes de tri sont parmi les plus étudiés
  - différents algorithmes
  - toujours un sujet de recherche
    - ✓ trouver le meilleur algorithme pour une certaine charge de travail
- Différentes idées peuvent être apprises via ces algorithmes
  - découpe en sous-problèmes
  - complexité
  - récursivité + diviser et régner
- Le tri est une composante fondamentale de l'informatique

Application (2)

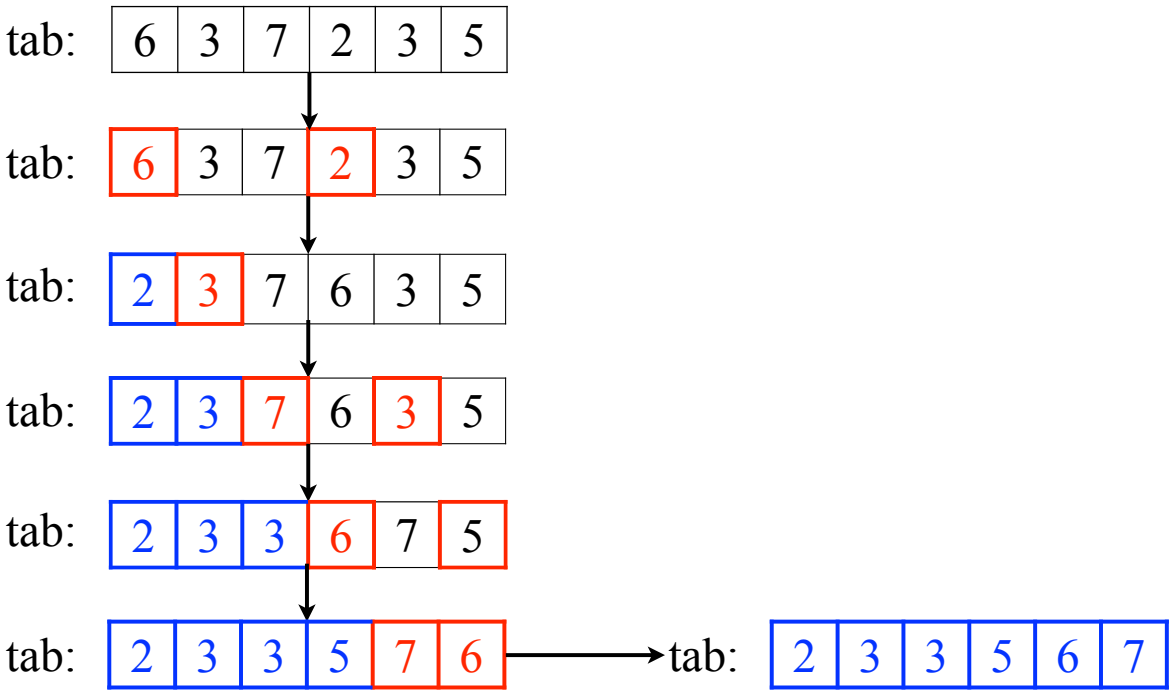
- Applications du tri
  1. recherche
     - ✓ recherche dichotomique
  2. paire la plus proche
     - ✓ étant donné *n* nombres, trouver les paires
     - ✓ une fois le tri effectué, c'est un problème "évident"
  3. unicité d'un élément
     - ✓ étant donné *n* nombres dans un "ensemble", sont-ils unique ou existe-t-il des duplicatas?
  4. calcul de statistiques
     - ✓ distribution de fréquence
     - ✓ calcul de la médiane (et autres quantiles)
  5. etc.

Application (3)

- Algorithme de tri
  - tri par sélection de valeurs
  - *Selection Sort*
- Principe
  - soit un tableau `tab` de *n* entiers
  - on cherche le minimum de tous les éléments et on le place en 1ère position dans le tableau
    - ✓ il reste à trier *n-1* éléments
  - on prend le minimum sur les *n-1* éléments restants et on le place en 2ème position dans le tableau
    - ✓ il reste à trier *n-2* éléments
  - et ainsi de suite

Application (4)

- Illustration du principe de l'algorithme

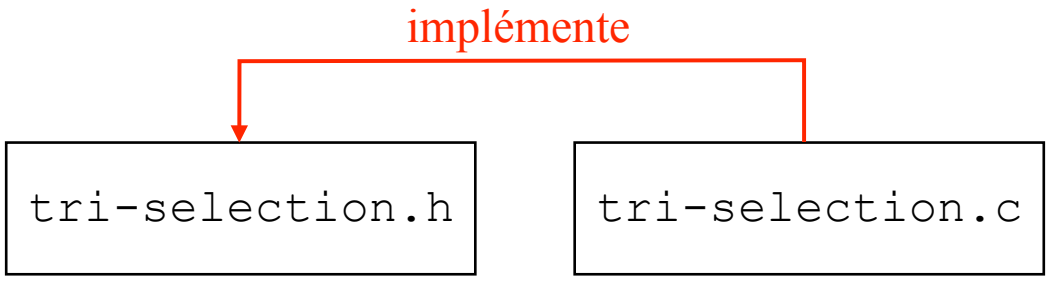


Application (5)

- Définition du problème
  - Input
    - ✓ `tab`, tableau d'entiers
    - ✓ `n`, taille du tableau
  - Output
    - ✓ le tableau est trié par ordre croissant
  - Objets Utilisés
    - ✓ $n \in \mathbb{N}_0$
    - ✓ le tableau `tab` existe et contient `n` valeurs entières

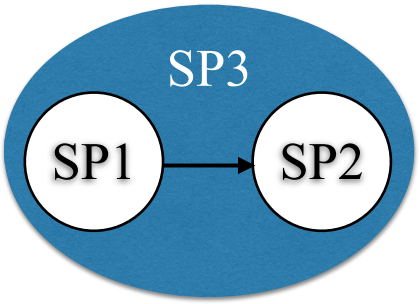
Application (6)

- Analyse du problème
  - architecture générale du code



Application (7)

- Analyse du problème (cont.)
  - SP1: retrouver l'indice du minimum dans un sous-tableau
  - SP2: permuter 2 éléments du tableau
  - SP3: problème général (i.e., tri)
- Interaction entre les SPs
  - (SP1 → SP2) ⊂ SP3



Application (8)

- Spécification du SP1 (indice du minimum)
  1. quels sont les paramètres?
     - ✓ un tableau, `tab`, de `n` valeurs entières
     - ✓ un indice, `debut`, indiquant le début du sous-tableau
  2. quel est l'objectif de ma fonction?
     - ✓ retourner la position du minimum dans `tab[debut ... n-1]`
     - ✓ postcondition: la position du minimum dans `tab[debut ... n-1]`
  3. quelles sont les contraintes sur les paramètres?
     - ✓ `n` ne peut être négatif ou nul
     - ✓ `tab` doit exister et contenir des valeurs
     - ✓ `debut` doit se trouver dans les bornes du tableau
     - ✓ `tab[0...debut-1]` déjà trié
     - ✓ précondition: `tab` initialisé, $0 \leq$ `debut` $<$ `n`, `n` > 0, `tab[0...debut-1]` trié ↗

Application (9)

- Spécification du SP2 (permutation)
  1. quels sont les paramètres?
     - ✓ un tableau, `tab`, de `n` valeurs entières
     - ✓ deux indices, `i` et `j`, indiquant les positions des éléments à permuter
  2. quel est l'objectif de ma fonction?
     - ✓ permuter les valeurs de `tab[i]` et `tab[j]`
     - ✓ postcondition: `tab[i]` <==> `tab[j]`
  3. quelles sont les contraintes sur les paramètres?
     - ✓ `n` ne peut être négatif ou nul
     - ✓ `tab` doit exister et contenir des valeurs
     - ✓ `i` et `j` doivent se trouver dans les bornes du tableau
     - ✓ précondition: `tab` initialisé, $0 \leq$ `i`, `j` $<$ `n`, `n` > 0

Application (10)

- Spécification du SP3 (tri)
  1. quels sont les paramètres?
     - ✓ un tableau, `tab`, de `n` valeurs entières
  2. quel est l'objectif de ma fonction?
     - ✓ trier par ordre croissant tab
     - ✓ postcondition: `tab` trié par ordre croissant
  3. quelles sont les contraintes sur les paramètres?
     - ✓ `n` ne peut être négatif ou nul
     - ✓ `tab` doit exister et contenir des valeurs
     - ✓ précondition: `tab` initialisé, `n` > 0

Application (11)

- Fichier `tri-selection.h`

```
/*
 * @pre: tab initialisé, 0 ≤debut< n, n>0, tab[0..debut-1] trié ↗
 * @post: minimum vaut la position du minimum dans
 *        tab[debut ... n-1]
 */
int minimum(int tab[], int n, int debut);

/*
 * @pre: tab initialisé, 0 ≤ i,j < n, n > 0
 * @post: tab[i] <==> tab[j]
 */
void permutation(int tab[], int n, int i, int j);

/*
 * @pre: tab initialisé, n > 0
 * @post: tab trié par ordre croissant
 */
void tri(int tab[], int n);
```


Application (12)

- Fichier `tri-selection.c`

```
#include <assert.h>
#include "tri-selection.h"
```

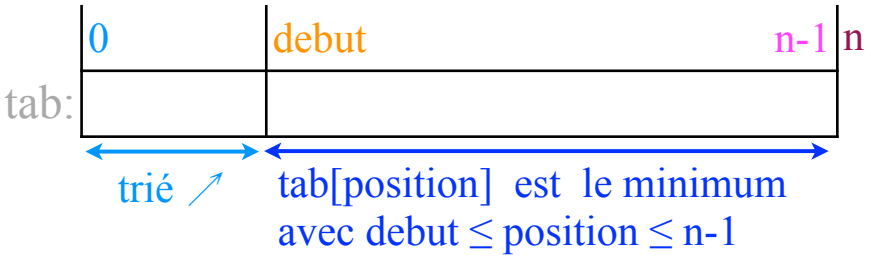
Application (13)

- Définition du SP1
  - Input
    - ✓ un tableau `tab` à `n` valeurs entières
    - ✓ `debut`, l'indice du début du sous-tableau
  - Output
    - ✓ la position du minimum dans `tab[debut ... n-1]`
  - Objets Utilisés
    - ✓ `tab` (tableau d'entiers), `n` (entier), `debut` (entier)
- Analyse
  - ∅
- Idée de solution
  - parcourir le tableau à partir de `debut` jusque `n-1`
  - maintenir le minimum "jusque maintenant" et sa position
  - retourner la position du minimum

Application (14)

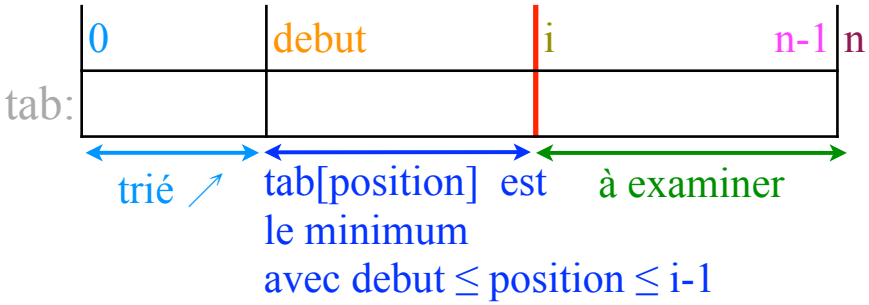
- Représentation graphique de l'Output

```
/*
 * @pre: tab initialisé, 0 ≤debut< n, n>0, tab[0..debut-1] trié ↗
 * @post: minimum vaut la position du minimum dans
 *.       tab[debut ... n-1]
 */
int minimum(int tab[], int n, int debut);
```



Application (15)

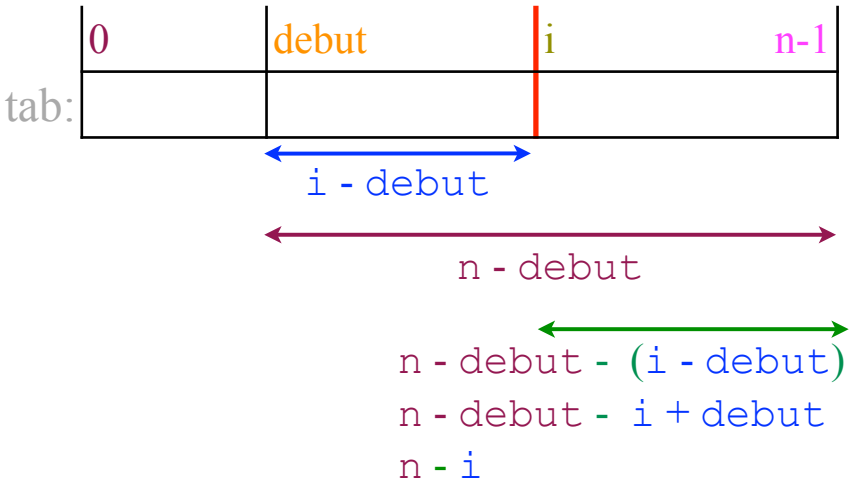
- Invariant Graphique pour le SP1



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Application (16)

- Construction de la fonction de terminaison

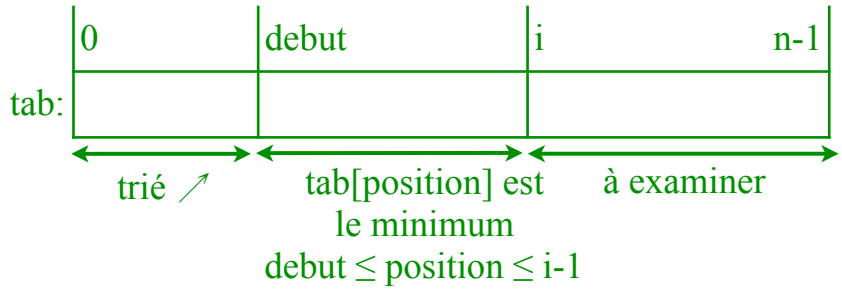


⇒ Fonction de Terminaison: n-i

Application (17)

- Code SP1

```
int minimum(int tab[], int n, int debut){
  assert(tab!=NULL && debut>=0 && debut<n && n>0);

  int position=debut, i = debut+1;

  while(i<n){
    if(tab[i]<tab[position])
      position = i;

    i++;
  }//fin while - i

  return position;
}//fin minimum()
```



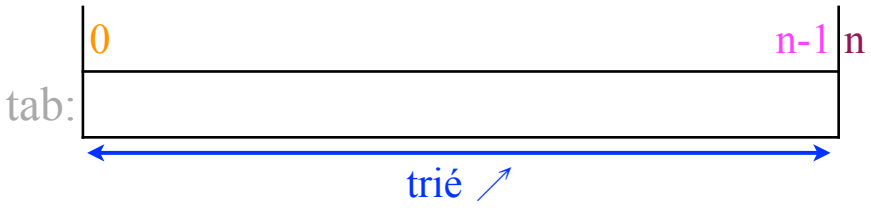
Application (18)

- Code SP2

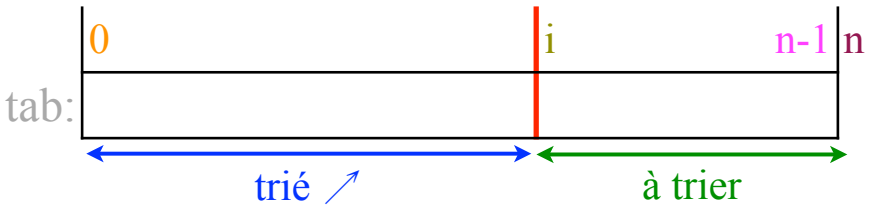
```
void permutation(int tab[], int n, int i, int j){
  assert(tab!=NULL && i>=0 && i<n && j>=0 && j<n && n>0);

  int tmp = tab[i];
  tab[i] = tab[j];
  tab[j] = tmp;
}//fin permutation()
```

Application (19)

- Représentation graphique de l'Output du SP3



- Invariant Graphique pour le SP3



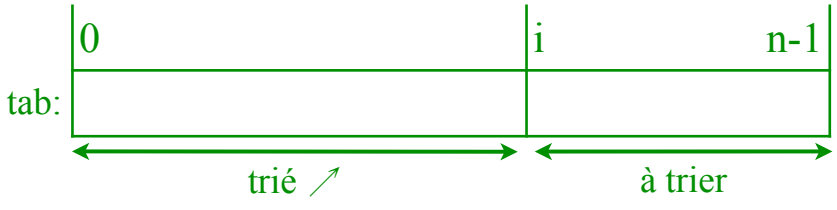
Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Fonction de Terminaison
  - n-i

Application (20)

- Code SP3

```
void tri(int tab[], int n){
  assert(tab!=NULL && n>0);
  int i, min_pos;

  for(i=0; i<n; i++){
     min_pos = minimum(tab, n, i);
     permutation(tab, n, i, min_pos);
  }//fin for - i
}//fin tri()
```



Application (21)

- Complexité du selection sort?

```
void tri(int tab[], int n){

  assert(tab!=NULL && n>0);
  int i = 0, min_pos;

  while(i<n){
     min_pos = minimum(tab, n, i);
     permutation(tab, n, i, min_pos);
     i++;
  }//fin for - i

}//fin tri()
```

T(A)

T(B')

T(B)

T(B'')

T(B''')

Application (22)

- Par application des règles 2 & 6
  - $T(n) = T(A) + T(B)$
- Par application des règles 1 & 5

$$T(n) = T(A) + T(B)$$

$$= 1 + \sum_{i=0}^{n-1} \left(T(B') + T(B'') + T(B''') \right)$$

$$= 1 + \sum_{i=0}^{n-1} \left(T(B') + 2 \right)$$

- Quid de T(B')?
  - évaluer la complexité de la fonction `minimum()`

Application (23)

- Complexité de `minimum()`

```
int minimum(int tab[], int n, int debut){
  assert(tab!=NULL && debut>=0 && debut<n && n>0);
  int position=debut, i = debut+1;
  while(i<n){
    if(tab[i]<tab[position])
      position = i;
    i++;
  }//fin while - i
  return position;
}//fin minimum()
```

T(A)

T(B')

T(B)

T(B")

T(C)

Application (24)

- Complexité de `minimum()` (cont.)

$$T(n) = T(A) + T(B) + T(C)$$

$$= 1 + \sum_{i=debut}^{n-1} \left(T(B') + T(B'')\right) + 1$$

$$= 2 + \sum_{i=debut}^{n-1} (1+1)$$

$$= 2 + \sum_{i=debut}^{n-1} 2$$

$$= 2 + \sum_{i=0}^{n-1} 2$$

$$= 2 \times n + 2$$

Application (25)

- Retour à la complexité du selection sort
  - $T(B') = 2 \times n + 2$
- Il vient

$$T(n) = 1 + \sum_{i=0}^{n-1} (2 \times n + 2 + 2)$$
$$= 1 + \sum_{i=0}^{n-1} (2 \times n + 4)$$
$$= 1 + 2 \times n^2 + 4 \times n$$
$$= 2 \times n^2 + 4 \times n + 1$$

- Par quoi borner *T(n)*?
  - $O(n^2)$
  - **complexité quadratique**

Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
    - ✓ Principe
    - ✓ Application
  - Variable Globale
  - Macro

Principe

- A l'intérieur d'une fonction, on peut déclarer une variable comme **`static`**
- La variable sera alors allouée statiquement
  - l'initialisation de la variable se fait une et une seule fois
  - la variable est partagée entre tous les appels de la fonction
- Une variable **`static`** dans une fonction est partagée entre tous les appels

Principe (2)

- Il est aussi possible de déclarer une fonction/ procédure avec le mot-clé **`static`**
- Intérêt?
  - la fonction/procédure ne sera visible/utilisable qu'à l'intérieur du module dans lequel elle est définie

Application

- Exemple

```c
#include <stdio.h>

int compteur(){
  static int x = 0;
  x++;

  return x;
}//fin compteur()

int main(){
  int i;

  for(i=0; i<3; i++)
    printf("%d\n", compteur());

  return 0;
}//fin programme
```


Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
    - ✓ Principe
    - ✓ Application
  - Macro

Principe

- Jusqu'à maintenant, on a considéré uniquement des variables définies dans un bloc
  - **variables locales**
- Il est possible, en C, de définir des variables en dehors de toute fonction (ou bloc d'instructions)
  - **variables globales**

type de la variable globale — identificateur de la variable globale

```
[static][const] type identificateur [= valeur]
                [, identificateur [= valeur]]
                                ...
                [, identificateur [=valeur]];
```

limitation de la portée au fichier de définition de la variable

Principe (2)

- La portée de la variable globale s'étend à tout le programme
  - même si le code est réparti dans plusieurs fichiers
- Pour utiliser une telle variable globale, il faut aider le compilateur à connaître son nom et son type

```
extern type identificateur [,identificateur [,...]];
```

Mot-clé indiquant que la variable globale est définie dans un autre fichier source

Principe (3)

- Les variables globales doivent utilisées avec prudence
  - effets de bord
    - ✓ les fonctions peuvent manipuler et modifier les variables globales
    - ✓ risque d'inconsistance
  - conflits de nom
    - ✓ si pas **`static`**, il peut y avoir des conflits de nom dans les grands programmes

Application

- Exemple: Compteur simple
  - fichier `compteur.h`

```
extern int compteur_valeur;

/*
 * @pre: -
 * @post: compteur_valeur initialisé à 0
 */
void compteur_init(void);

/*
 * @pre: compteur_valeur ≥ 0
 * @post: le compteur est incrémenté d'une unité
 */
void compteur_plus(void);
```

Application (2)

- Exemple: Compteur simple
  - fichier `compteur.h` (suite)

```
/*
 * @pre: compteur_valeur ≥ 0
 * @post: compteur_valeur décrémenté de 1 unité &
 *        compteur_valeur ≥ 0
 */
void compteur_moins(void);

/*
 * @pre: compteur_valeur ≥ 0
 * @post: (compteur_valeur==0)
 */
int compteur_est_zero(void);
```


Application (3)

- Fichier `compteur.c`

```
#include "compteur.h"
int compteur_valeur;

void compteur_init(){
  compteur_valeur = 0;
}//fin compteur_init()

void compteur_plus(){
  compteur_valeur++;
}//fin compteur_plus()

void compteur_moins(){
  if(compteur_valeur>0)
    compteur_valeur--;
}//fin compteur_moins()
```

```
int compteur_est_zero(){
  return !compteur_valeur;
}//fin compteur_est_zero()
```

Application (4)

- Fichier `compteur-main.c`

```c
#include <stdio.h>
#include "compteur.h"

static void affiche(){
  printf(compteur_est_zero() ? "== 0\n" : "!= 0\n");
}//fin affiche()

int main(){
  compteur_init(); compteur_plus(); compteur_plus();
  compteur_moins();
  affiche();
  compteur_moins();
  affiche();

  return 0;
}//fin programme
```


Agenda

- Chapitre 6: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Programmation par Contrat
  - Variable Statique
  - Variable Globale
  - Macro

Macro

- Il existe, en C, un mécanisme (autre que la fonction) permettant de déployer un même fragment de code à plusieurs endroits
  - **macro**
- Directive de pré-traitement
  - il s'agit d'une substitution syntaxique
  - appliquée avant la compilation
- Construction

```
#define nom [(id1 [,id2 [,...]])] texte
```

Paramètres éventuels

Directive de pré-traitement

toute occurrence de "nom" sera remplacée par "texte" dans la suite du programme

Macro (2)

- Exemples

```
#define PI 3.1415926535897932384

#define square(x) ((x) * (x))

#define max(a, b) ((a) > (b) ? (a) : (b))
```

- Les macros ne sont pas des fonctions!
  - le comportement diffère quand l'évaluation des arguments provoque des effets de bords
    - ✓ `square(n++)` se substitue en `((n++) * (n++))`
    - ✓ `n` incrémenté deux fois
  - la substitution se fait vraiment de manière textuelle
    - ✓ `#define N = 5`
    - ✓ `int t[N]` ⇒ `int t[= 5]`

Macro (3)

- Attention à la priorité des opérateurs
  - `#define square(x) x * x`
  - `square(a+b)`
    - ✓ `a+b * a+b`
  - ce n'est pas ce que l'on souhaite
- Emploi des parenthèses conseillé
- Globalement, à utiliser avec beaucoup de prudence