Introduction à l'Informatique

Benoit Donnet

Année Académique 2017 - 2018

Agenda

- Introduction
- Chapitre 1: Bloc, Variable, Instruction Simple
- Chapitre 2: Structures de Contrôle
- Chapitre 3: Méthodologie
- Chapitre 4: Structures de Données
- Chapitre 5: Modularité du Code
- Chapitre 6: Pointeurs

Agenda

- Chapitre 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Spécifications
  - Macro

Agenda

- Chapitre 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Spécifications
  - 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
      - ‣ cfr. Compilation Séparée

Principe (4)

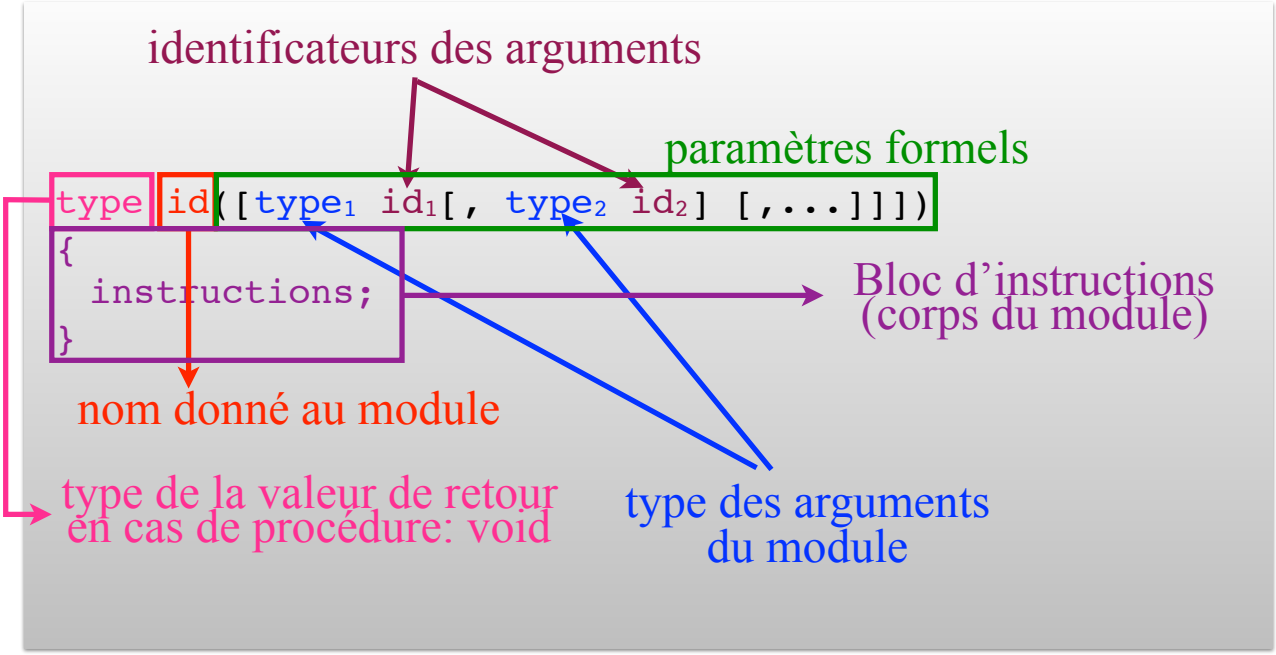
- Fonctionnement?
  - le code invoque un module
    - ✓ celui qui invoque est appelé **code appelant**
  - pendant l'exécution du module, l'exécution du code appelant est suspendu
    - ✓ c'est le module qui a la main
  - le code appelant reprend son exécution lorsque le module invoqué est terminé
- 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. 6 pour le détail sur le *passage de paramètres*

Agenda

- Chapitre 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
    - ✓ Déclaration
    - ✓ Portée des Variables
    - ✓ Utilisation
    - ✓ Application
  - Compilation Séparée
  - Spécifications
  - 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
  - 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

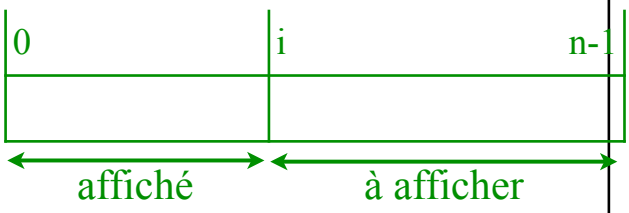
identificateur

paramètres formels

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

variable locale

//Inv: tab:



Déclaration (4)

- Exemple 2
  - fonction qui retourne x^7

retourne une valeur de type `int`

identificateur

paramètre formel

```
int puissance7(int x){

  int resultat;

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

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

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()`

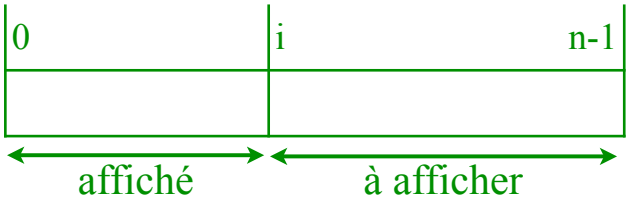
Déclaration (7)

- 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

Utilisation

- Comment utiliser 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

Utilisation (2)

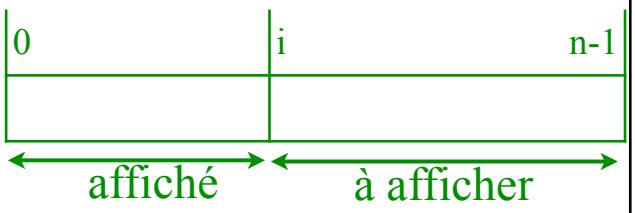
- Si `m` est une fonction
  - `m` est une valeur à droite
    - ✓ peut se trouver à droite d'une affectation
  - `m` est vu comme une expression
    - ✓ dont l'évaluation est égale à la valeur retournée par `m` à la fin de son exécution
  - exemple
    - ✓ utilisation de `fopen()`
    - ✓ utilisation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

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

//Inv: tab:

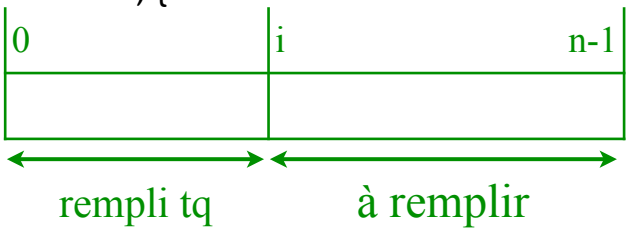


Utilisation (4)

- Exemple (suite)

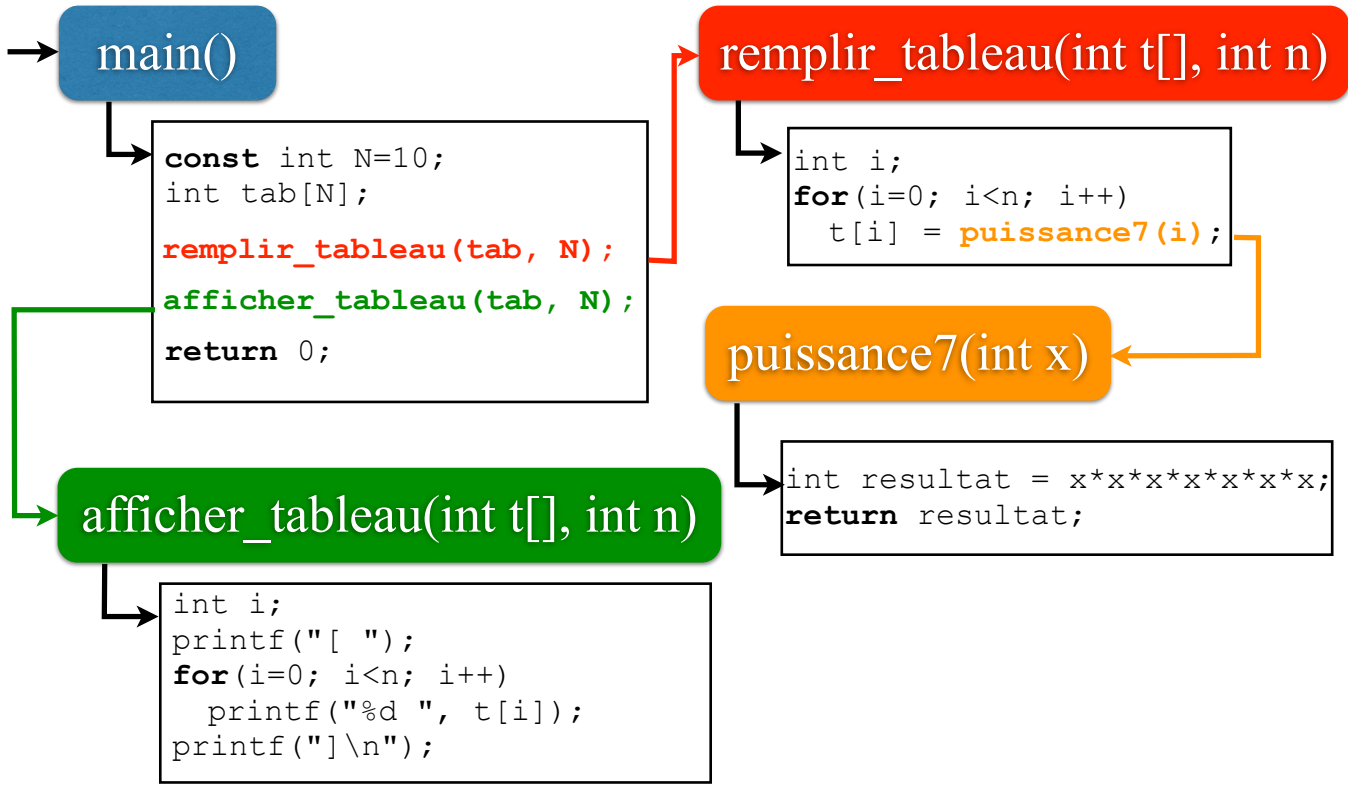
```
#include <stdio.h>

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

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

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

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

  return 0;
}//fin programme
```

//Inv: t:



$t[j] = j^7, \forall\ j, 0 \leq j \leq i-1$

Utilisation (5)



Application

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

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

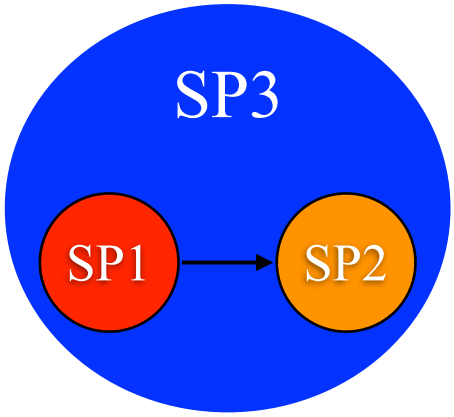
Application (2)

- 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!

```
#include <stdio.h>

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

  //Inv: fact = (i-1)!, 1 ≤ i ≤ n+1
  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

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

  //Inv: exp=a^(i-1), 1 ≤ i ≤ b+1
  for(i=1; i<=b; i++)
    exp *= a;

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

Application (6)

- 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)

- Problème général

```
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 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
    - ✓ Principe
    - ✓ Application
  - Spécifications
  - 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
  - `source.c`
- Le header et fichier d'implémentation ont le même nom
  - le fichier d'implémentation inclut le header associé
  - et tous les headers nécessaires

Application

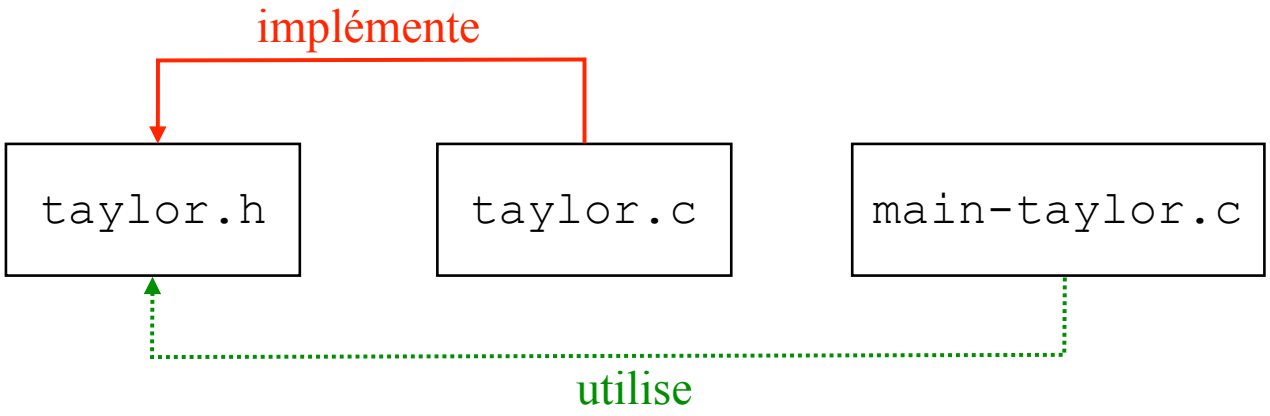
- Application:
  - approximation de *sinus(x)* et *cosinus(x)* par un développement en série de Taylor

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

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

Application (2)

- Architecture du code



Application (3)

- Fichier `taylor.h`

```
// 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)

- Fichier `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)

- Fichier `taylor.c` (cont.)

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

  //Inv: exp=a^(i-1), 1 ≤ i ≤ b+1
  for(i=1; i<=b; i++)
    exp *= a;

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

Application (6)

- Fichier `taylor.c` (cont.)

```
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} (-1)^i/(2i+1)! x^{2i+1}, 0 <= n <= precision
  return sin;
}//fin sinus_taylor()
```


Application (7)

- Fichier `taylor.c` (cont.)

```
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} (-1)^i/(2i)! x^{2i}, 0 <= n <= precision
  return cos;
}//fin cosinus_taylor()
```

Application (8)

- Fichier `main-taylor.c`

```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 (8)

- 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 impôt 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 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Spécifications
    - ✓ Principe
    - ✓ Programmation Défensive
    - ✓ Application
  - Macro

Principe

- 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

Principe (2)

- La description du travail et les contraintes d'utilisation sont expliquées grâce à des **spécifications**
  - contrat logiciel qui lie
    - le programmeur de la fonction
    - l'utilisateur de la fonction
- 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
  - **Postcondition**
    - caractérise les conditions finales d'exécution du module
    - en particulier le résultat
    - sera satisfaite après l'appel

Principe (3)

- Lien avec la méthodologie de résolution de problèmes
  - les spécifications concernent la définition d'un problème
    - précondition == données en entrée
    - postcondition == résultat(s) attendu(s) + forme
- Lien avec les invariants
  - la précondition doit amener l'invariant
    - Pré ⇒ Inv
  - l'invariant et la sortie de la boucle doivent amener la postcondition
    - Inv && !cond ⇒ Post

Principe (4)

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

Principe (5)

- Comment écrire correctement une fonction et sa spécification?
- Il y a trois questions à se poser
  1. quel est l'objectif de ma fonction?
     - ✓ post-condition
  2. quels sont les paramètres (nom, type, signification) nécessaires pour que la fonction puisse réaliser cet objectif?
     - ✓ prototype de la fonction
  3. quelles sont les contraintes sur ces paramètres?
     - ✓ pré-condition

Principe (6)

- Exemple 1
  - fonction qui retourne le minimum d'un tableau d'entiers
- Trois questions
  1. quels sont les paramètres?
     - ✓ un tableau `tab` de `n` valeurs entières
  2. quel est l'objectif de la fonction?
     - ✓ retourner le minimum de `tab`
     - ✓ postcondition: min{`tab`[0], `tab`[1], ..., `tab`[n-1]}
  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

Principe (7)

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

```
/*
 * @pre: tab est initialisé, n > 0
 * @post: MIN{tab[0], tab[1], ..., tab[n-1]}
 */
int minimum(int tab[], int n){
  int min = tab[0], i;
  //Inv: min = MIN{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 = MIN{tab[0], ..., tab[i-1]}, 1 ≤ i ≤ n && i≥n
  //⇒ min=MIN{tab[0], ..., tab[n-1]}
  return min;
}//fin minimum()
```

Principe (8)

- Exemple 2
  - 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

Principe (9)

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

/*
 * @pre: b ≥ 0
 * @post: 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: calcul de factorielle

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

```
/*
 * @pre: n ≥ 0
 * @post: n!
 */
int factorielle(int n){
  assert(n>=0);
```
vérifie la précondition sur `n`

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

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

Progra. Défensive (5)

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

  return 0;
}//fin programme
```

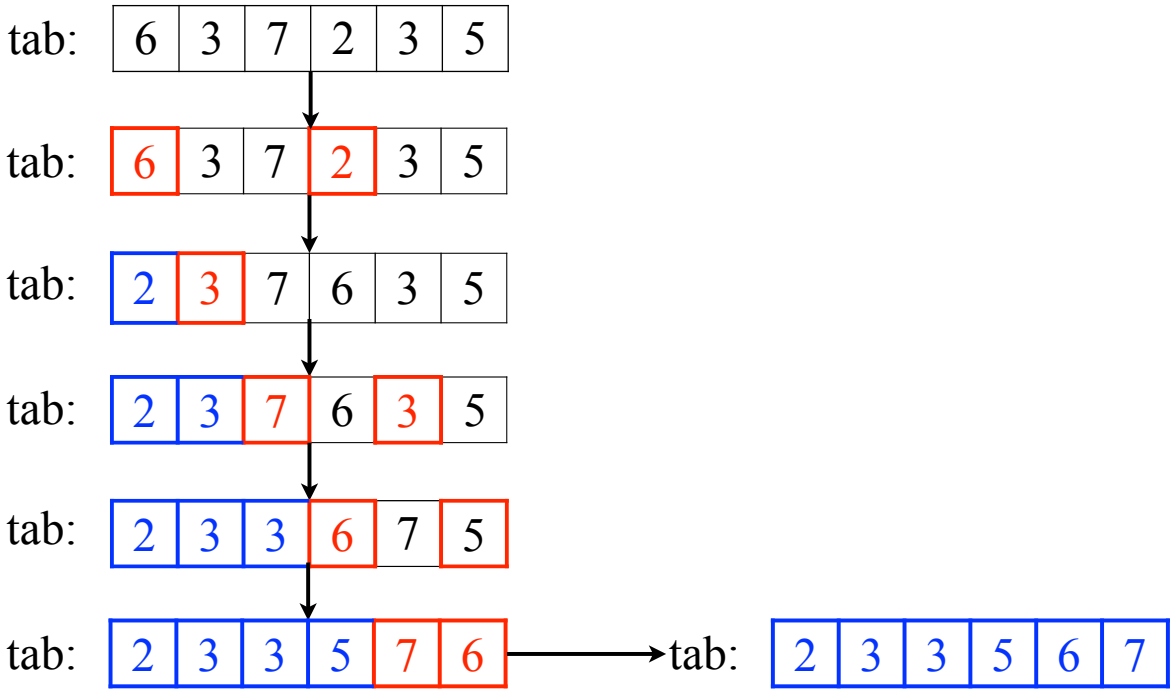
```
$> gcc -o factorielle factorielle.c
$>./factorielle
120
1
Assertion failed: (n>=0), function factorielle, file
factorielle.c, line 9.
Abort trap: 6
```


Application

- Application
  - tri d'un tableau d'entiers par sélection
  - *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 restant et on le place en 2ème position dans le tableau
    - ✓ il reste à trier *n-2* éléments
  - et ainsi de suite

Application (2)

- Illustration du principe de l'algorithme

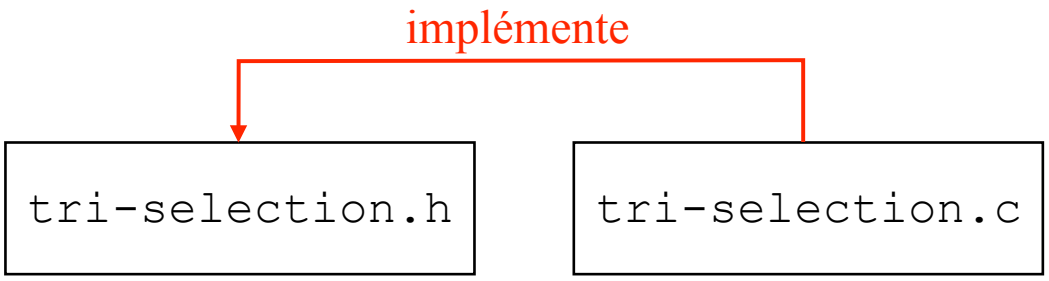


Application (3)

- 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

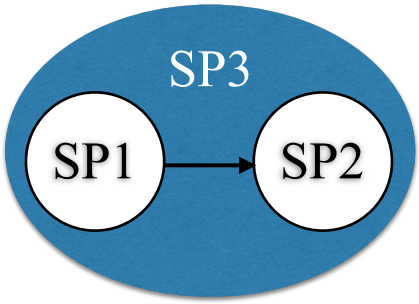
Application (4)

- Analyse du problème
  - architecture générale du code



Application (5)

- 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 (6)

- 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
     - ✓ précondition: `tab` initialisé, $0 \leq$ `debut` $<$ `n`, `n` > 0

Application (7)

- 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 (8)

- 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 (9)

- Fichier `tri-selection.h`

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

/*
 * @pre: tab initialisé et 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 (10)

- Fichier `tri-selection.c`

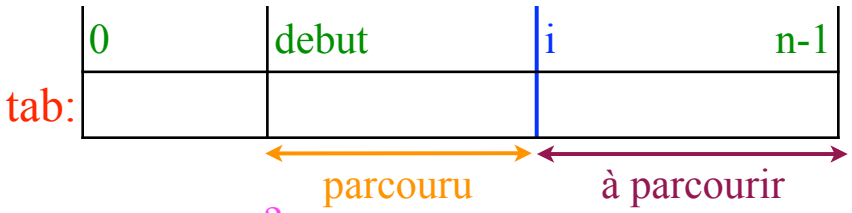
```
#include <assert.h>
#include "tri-selection.h"
```


Application (11)

- 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 (12)

- Invariant graphique pour le SP1



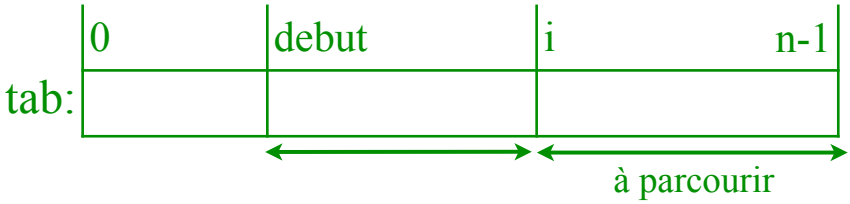
parcouru?
tab[position] = MIN{tab[debut], ..., tab[i-1]}
avec debut ≤ position ≤ i-1

- Invariant en français pour le SP1:
  - `tab[position]` contient le minimum dans `tab[debut ... i-1]`, avec $0 \leq$ `debut` $<$ `i` $\leq$ `n` et `debut` $\leq$ `position` $\leq$ `i-1`

Application (13)

- 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()
```



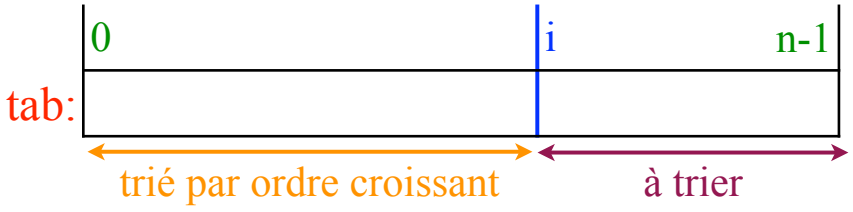
tab[position] = MIN{tab[debut], ..., tab[i-1]}
debut ≤ position ≤ i-1

Application (14)

```
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 (15)

- Invariant graphique pour le SP3

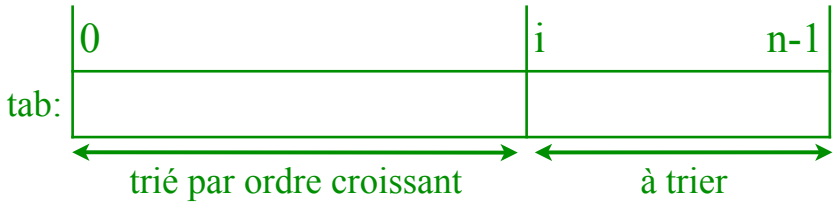


- Invariant en français pour le SP3:
  - `tab[0 ... i-1]` est trié par ordre croissant, avec $0 \leq i \leq n$

Application (16)

- 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()
```



Agenda

- Chapitre 5: Modularité du Code
  - Principe
  - Fonctions et Procédures
  - Compilation Séparée
  - Spécifications
  - 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 3.14159265358979323 8

#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