Programmation C

Variables et Fonctions

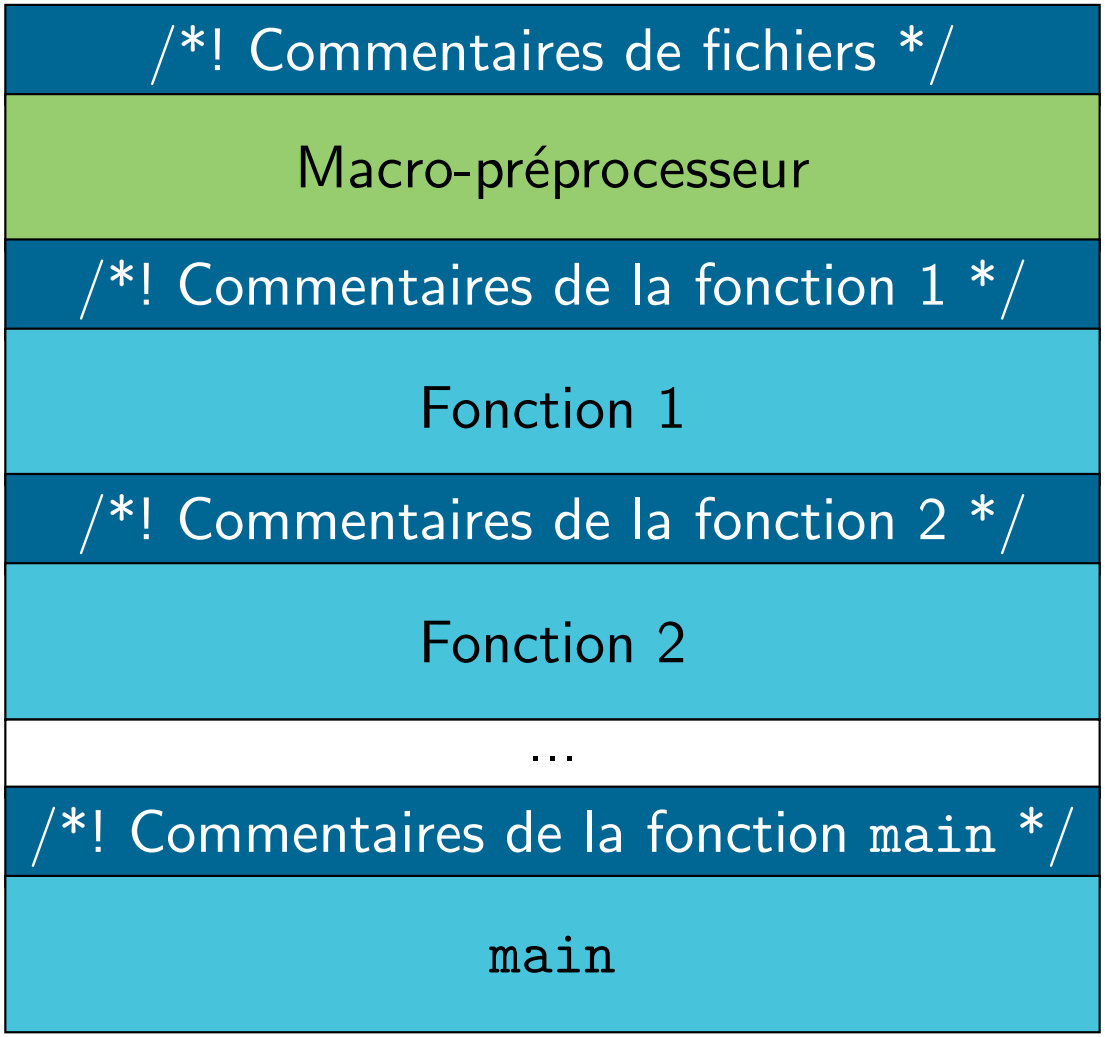
ING1

CY Tech

Fonctions

Rappel

Structure d'un fichier `.c`



Prototype

- type de retour
- nom de la fonction
- type des paramètres
- éventuellement le nom des paramètres
- se termine par un ; $\Leftrightarrow$ n'inclut pas le corps de la fonction
- permet de savoir comment utiliser la fonction

```
int abs(int);
int labs(long int);

int printf (const char *format, ...);
```

- déclare une fonction

Paramètres

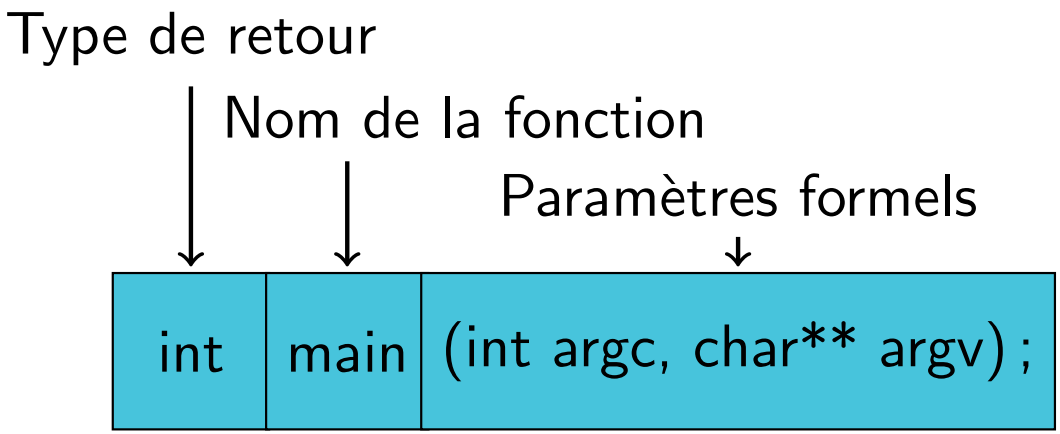


Figure – Exemple de prototype

Paramètres

Formel ou Effectif

- Formel
  - Nom du paramètre
  - Utile pour la déclaration
- Effectif
  - Valeur du paramètre
  - Uniquement lors de l'exécution

Mécanisme d'appel

Passage par valeur (ou par copie)

- Évaluation des paramètres appelant
- Copie de la valeur de ceux-ci à la fonction
- Exécution de la fonction
- Retour de la valeur obtenue

Contexte d'exécution

- Ce que connaît la fonction
  - ses paramètres
  - les variables globales
  - les variables locales
- Ne connaît pas les variables des autres fonctions

Fonctions, procédures ?

procedure afficheCarres(debut, fin: entier)
.../...
fin procedure
fonction calculCarres(nb: entier): entier
.../...
fin fonction

```
void afficheCarres (int int_debut, int int_fin) {
  .../...
}
int calculCarres (int int_nb) {
  .../...
  return (...);
}
```

Attention

Ordre de définition des fonctions

Une fonction ne connait que les fonctions définies **AVANT** elle.

Variables

Types de variables

```c
#include <stdio.h>

int Gint_global ; // Variable globale

int main (int argc, char** argv) {
   int int_local; // Variable locale
   .../...
}
```

Une variable a la durée de vie de son bloc

Types de variables

Types de variables

- Globales
  - À déclarer entre le début du fichier et la première fonction. Idéalement entre les macros-préprocesseurs et le début
  - Ne correspond pas à la déclaration de variable dans le *Programme*
  - Interdit d'utilisation à CY Tech, sauf obligation
- Locales
  - Locale à la fonction
  - Durée de vie limitée $\Leftrightarrow$ n'existe que pendant le temps d'exécution de la fonction

Qualificatif

- `const` : variable ne doit pas être modifiable
  - généralement pour les paramètres de méthodes en "lecture seule"
- `volatile` : variable peut être modifiée par d'autres programmes
  - variable partagée par d'autres programmes
  - zone de mémoire partagée
  - ...
- `register` : variable doit être placée dans un registre processeur (variable utilisée fréquement)
- `extern` : variable définie ailleurs
- `static` : permet d'avoir une variable avec une durée de vie de la durée du programme

Test de fonction

Test

Fonction

- Doit tester ses paramètres
- Utilisation des assertions peut être utile pendant la phase de codage [a]
- Utilisation de test et de code d'erreur
- Définition de constante pour une meilleure lisibilité

---

a. Attention aux effets de bord !

Constantes symboliques

- Remplacer une "expression" par un symbole dans le code
- Permet de clarifier le code
- `#define symbole expression`
  - `#define N 10`
  - `#define ERREUR_SAISIE -1`
  - `#define M 20;`
- Utilisation des majuscules
- Éviter de mettre un ; à la fin d'un `define`

Exemple

```c
#include <stdio.h>
#include <stdlib.h>
#define ERREUR_SAISIE -1

int main (int argc, char** argv) {
   .../...
   int_retour = scanf("%d", &int_val);
   // test de la lecture
   if (int_retour == 0) {
      .../...
      /* on quitte avec le code d'erreur
      associe a la saisie */
      exit(ERREUR_SAISIE);
   }
   .../...
}
```

Doxygen

```
/*!
   \def ERREUR_SAISIE
   Code d'erreur associe a la saisie
*/
#define ERREUR_SAISIE -1
```

Assertion

```
int divise (int int_a, int int_b) {
   assert(int_b != 0);
   return(int_a / int_b);
}
```

- Utile lors de la phase de codage
- Peut être enlevé en utilisant `-DNDEBUG` lors de la compilation

Test de paramètres et autres

- TOUS les paramètres d'une fonction devraient être testés, sauf si la fonction admet un domaine de définition infini
- Deux cas peuvent
  - valeur en dehors de la plage d'utilisation mais acceptable $\Leftrightarrow$ utilisation de test et impression de message sur la sortie d'erreur standard (`fprintf` et `strerror`)
  - valeur inacceptable $\Leftrightarrow$ utilisation d'`exit` pour quitter le programme
- Les valeurs de retour des fonctions doivent être testées, sauf si l'on est sûr qu'il n'y aura JAMAIS d'erreur

Exemple

```
int_retour = scanf("%d", &int_val);
/* tant que la saisie est incorrecte,
on redemande une saisie */
while (int_retour == 0) {
   fprintf(stderr, "Entree incorrecte \n");
   printf("Entrer une autre valeur : ");
   // Attention aux problemes de buffer !
   int_retour = scanf("%d", &int_val);
}
assert(int_val < 0);
if (int_val < 0) {
   exit(ERREUR_UTILISATEUR);
}
```

Traitement des erreurs

- Besoin de détecter les erreurs possibles
- Indiquer ce qui ne va pas
- Donner des messages clairs !
- Utiliser le bon canal de communication $\Leftrightarrow$ `stderr`

Utilisation de `fprintf`

- Permet de spécifier un fichier de sortie
- Unix : trois fichiers ouverts par défaut
  - entrée standard : `stdin`
  - sortie standard : `stdout`
  - sortie d'erreur standard : `stderr`
- S'utilise comme `printf`

Exemple

```
int_retour = scanf("%d", &int_val);
// Test de la saisie
if (int_retour == 0) {
   fprintf(stderr, "Entree incorrecte \n");
   .../...
}
```

Utilisation de `strerror`

- Possibilité de "remonter" l'erreur système
- Utilisation de `strerror` avec `errno`
  - `errno` : positionné automatiquement par les fonctions
  - utilisation uniquement quand possible ...
  - ouverture de fichiers, copie, ...

```
fprintf(stderr, "%s \n", strerror(errno));
```