I3 – Algorithmique et programmation
Introduction à l'algorithmique

Cours n°3
Programmation structurée
Programmation en C : premier programme

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2011 – 2012

Plan

Décomposition d'un problème en sous-problèmes

"... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre."

Discours de la méthode (1637)
René Descartes

Fonctions

Définition

Une **fonction** définit une action effectuée sur un ensemble de paramètres qui produit un résultat retourné comme valeur de sortie de la fonction.
Elle fournit une **abstraction** pour cette action vis-à-vis des algorithmes qui l'appellent.

Analogie avec les fonctions mathématiques : $f : x \mapsto f(x)$

- On définit la fonction une fois
- On l'appelle autant de fois qu'on veut

Avantages : factorisation de code, abstraction pour la conception de l'algorithme... Permet de se concentrer sur l'algorithme lui-même plutôt que sur les détails.

Paramètres et résultat

On donne les paramètres et leur type

- On déclare les paramètres entre parenthèses
- On les utilise dans la fonction en utilisant **ce nom** : ce sont des variables
- Il existe des paramètres
  - D'entrée : utilisés dans la fonction
  - De sortie : modifiés dans la fonction
  - Ou les deux : utilisés et modifiés dans la fonction

On déclare le type retourné

- Déclaration de la fonction suivie par **: type**
- On sort de la fonction avec le mot clé **retourner** et la valeur retournée

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

Appel de fonction

On appelle la fonction à partir d'un autre algorithme

- On lui passe en paramètres des variables de l'algorithme appelant
- La valeur retournée est mise dans une variable de l'algorithme appelant

```
1 début programme
2 |   nombreDepart : Entier
3 |   calcul : Entier
4 |   calcul ⟵ carre ( nombreDepart )
5 fin programme
```

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

La fonction fournit donc une **abstraction** de l'action qu'elle réalise paramétrée par les variables passées lors de l'appel.

Procédures

Une procédure effectue une **action**. Elle ne retourne rien.

- Affichage : pas de résultat de calcul à récupérer
- Calcul sur des **variables de sortie**

On sort de la procédure

- À la fin du bloc d'instructions principal
- Ou quand on rencontre le mot-clé **Retour**

Procédures (exemple)

```
1 début programme
2 |   mapuissance : Entier
3 |   puissance ( mapuissance )
4 |   afficher ( mapuissance )
5 fin programme
```

```
1 début procédure puissance( petitepuissance: Entier Sortie )
2 |   petitepuissance ⟵ 1
3 |   tant que Vrai faire
4 |   |   petitepuissance ⟵ petitepuissance × 2
5 |   |   if petitepuissance > 200 then
6 |   |   |   retourner
7 |   |   endif
8 |   fintq
9 fin procédure
```

Visibilité des variables

Les variables déclarées dans la fonction

- ne sont visibles **que** dans la fonction

Les variables déclarées dans le programme appelant

- ne sont visibles **que** dans le programme appelant
- ne sont **pas** visibles dans la fonction

Les variables passées en paramètre

- sont appelées par leur nom dans le programme appelant dans l'appel de la fonction
- sont appelées par le nom utilisé pour les déclarer dans la fonction elle-même

Visibilité des variables

Les variables sont visibles uniquement dans la fonction, la procédure ou le programme dans lequel elles ont été déclarées, et dans aucune des fonctions ou procédures appelées ou qui l'appellent.

Approche descendante

Les fonctions et les procédures fournissent une **abstraction sur les actions réalisées par le programme**

- Possibilité de les utiliser pour se concentrer sur la structure du programme plutôt que sur la résolution des sous-problèmes

On décompose le problème en **sous-problèmes**

- On fait dans un premier temps l'hypothèse qu'ils sont résolus
- On s'en occupe plus tard

La conception de l'algorithme dans sa globalité est ainsi **plus simple**

- Décomposer pour mieux maîtriser

On retarde le plus possible le moment d'effectuer les calculs

Approche descendante : exemple

Théorème de Pythagore : *"le carré de l'hypoténuse d'un triangle rectangle est égal à la somme des carrés des longueurs des deux autres côtés"*.

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2      /* variables internes */
3      carre1 : Entier
4      carre2 : Entier
5      hypotenusecarre : Entier
6      hypotenuse : Entier
7      /* on calcule la somme des carrés des côtés */
8      carre1 ⟵ carré( cote1 )
9      carre2 ⟵ carré( cote2 )
10     hypotenusecarre ⟵ carre1 + carre2
11     /* on prend la racine carrée de la somme et on retourne le
       résultat */
12     hypotenuse ⟵ racine( hypotenusecarre )
13     retourner hypotenuse
14 fin fonction
```

Approche descendante : exemple (suite)

```
1 début fonction carré( nombre: Entier ): Entier
2 |   moncarre : Entier
3 |   moncarre ⟵ nombre × nombre
4 |   retourner moncarre
5 fin fonction
```

```
1 début fonction racine( nombre: Entier ): Entier
2 |   maracine : reel
3 |   maracine ⟵ √nombre
4 |   retourner maracine
5 fin fonction
```

Approche descendante : autre exemple

On veut écrire un procédure qui affiche une pyramide comme celle-ci de hauteur passée en paramètre :

```
      1
    1 2 1
  1 2 3 2 1
1 2 3 4 3 2 1
```

On l'affiche ligne par ligne :

```
1 début procedure afficherPyramide( hauteur: Entier )
2 |   pour i ⟵ 0 à hauteur pas 1 faire
3 |   |   afficherLigne( i, n )
4 |   finpour
5 fin procedure
```

Approche descendante : autre exemple

On veut écrire un procédure qui affiche une pyramide comme celle-ci de hauteur passée en paramètre :

```
      1
    1 2 1
  1 2 3 2 1
1 2 3 4 3 2 1
```

On l'affiche ligne par ligne :

```
1 début procedure afficherPyramide( hauteur: Entier )
2 |   pour i ⟵ 0 à hauteur pas 1 faire
3 |   |   afficherLigne( i, n )
4 |   finpour
5 fin procedure
```

Approche descendante : autre exemple (suite)

On donne la fonction `afficherLigne()` :

```
1 début procedure afficherLigne( largeur: Entier, hauteur : Entier )
2 |   afficherBlancs( i, n ) afficherSuiteCroissante( i )
  |   afficherSuiteDecroissante( i ) allerALaLigne()
3 fin procedure
```

Et on donne les fonctions appelées par `afficherLigne()` :

```
1 début procedure afficherBlancs( largeur: Entier, hauteur : Entier )
2 |   pour i ⟵ 0 à hauteur − largeur pas 1 faire
3 |   |   afficher( " " )
4 |   finpour
5 fin procedure
```

Approche descendante : autre exemple (suite)

On donne la fonction `afficherLigne()` :

```
1 début procedure afficherLigne( largeur: Entier, hauteur : Entier )
2 |   afficherBlancs( i, n ) afficherSuiteCroissante( i )
  |   afficherSuiteDecroissante( i ) allerALaLigne()
3 fin procedure
```

Et on donne les fonctions appelées par `afficherLigne()` :

```
1 début procedure afficherBlancs( largeur: Entier, hauteur : Entier )
2 |   pour i ⟵ 0 à hauteur − largeur pas 1 faire
3 |   |   afficher( " " )
4 |   finpour
5 fin procedure
```

Approche descendante : autre exemple (suite)

```
1 début procedure afficherSuiteCroissante( largeur: Entier )
2 |   pour i ⟵ 0 à largeur pas 1 faire
3 |   |   afficher( i )
4 |   finpour
5 fin procedure
```

```
1 début procedure afficherSuiteDecroissante( largeur: Entier )
2 |   pour i ⟵ largeur − 1 à 0 pas −1 faire
3 |   |   afficher( i )
4 |   finpour
5 fin procedure
```

```
1 début procedure allerALaLigne( )
2 |   afficher( retour_chariot )
3 fin procedure
```

Récursivité

Une fonction ou une procédure peut **s'appeler elle-même**

- Elle est alors partiellement définie à partir d'elle-même

```
1 début fonction calcul( var: Entier ): Entier
2 |   si var == 1 alors
3 |   |   retourner var
4 |   sinon
5 |   |   retourner var×calcul( var − 1 )
6 |   finsi
7 fin fonction
```

- Correspond bien aux relations de récurrence
- Attention au point d'arrêt !

Éditeur de texte

Utilisation d'un **éditeur de texte sans formattage**

- Écriture du texte brut dans le fichier
- Pas d'informations de formattage

Exemples : Emacs, vi(m), gedit...

Possibilité d'utiliser un Environnement de Développement Intégré (IDE)

- Éditeur de texte
- Compilation
- Exécution
- Débuggage
- Accès à l'aide en ligne et raccourcis

Exemples : Eclipse, NetBeans, Visual Studio...

Compilation

Langage C = code compréhensible par un humaine
Exécutable = langage machine compréhensible par la machine

Compilation

La **compilation** consiste à traduire un langage de programmation dans un autre langage. Le langage cible est souvent le langage machine d'une machine sur laquelle le programme va être exécuté, mais il peut s'agir d'un autre langage (compilateur source-to-source).

Le langage C doit être **compilé** pour être exécuté sur une machine.
Exemples de compilateurs C : gcc (GNU), icc (Intel), pathcc (Pathscale), xlcc (IBM)...

Chaîne de production d'un exécutable

1. **Pré-processing** :
   - inclusion des en-têtes
   - suppression des commentaires
   - remplacement des macros et des constantes

   Code produit : code source modifié, en langage C, lisible par un être humain.
2. **Compilation** : analyse du respect de la syntaxe du langage et transformation en **code objet**. Trois phases :
   - **Analyse lexicale et syntaxique** du fichier source
   - **Génération** d'un code en langage d'assemblage
   - **Traduction du code** en langage d'assemblage vers un code objet

   Code produit : langage machine, plusieurs fichiers (1 par fichier source)
3. **Édition des liens**
   - Si le programme est constitué de plusieurs fichiers source et/ou si il utilise des fonctions et des variables externes (définies dans des bibliothèques extérieures), les bouts de code objet correspondants sont reliés entre eux pour former un **exécutable**.

   Code produit : langage machine, un exécutable

Chaîne de production d'un exécutable

1. **Pré-processing** :
   - inclusion des en-têtes
   - suppression des commentaires
   - remplacement des macros et des constantes

   Code produit : code source modifié, en langage C, lisible par un être humain.
2. **Compilation** : analyse du respect de la syntaxe du langage et transformation en **code objet**. Trois phases :
   - **Analyse lexicale et syntaxique** du fichier source
   - **Génération** d'un code en langage d'assemblage
   - **Traduction du code** en langage d'assemblage vers un code objet

   Code produit : langage machine, plusieurs fichiers (1 par fichier source)
3. **Édition des liens**
   - Si le programme est constitué de plusieurs fichiers source et/ou si il utilise des fonctions et des variables externes (définies dans des bibliothèques extérieures), les bouts de code objet correspondants sont reliés entre eux pour former un **exécutable**.

   Code produit : langage machine, un exécutable

Chaîne de production d'un exécutable

1. **Pré-processing** :
   - inclusion des en-têtes
   - suppression des commentaires
   - remplacement des macros et des constantes

   Code produit : code source modifié, en langage C, lisible par un être humain.
2. **Compilation** : analyse du respect de la syntaxe du langage et transformation en **code objet**. Trois phases :
   - **Analyse lexicale et syntaxique** du fichier source
   - **Génération** d'un code en langage d'assemblage
   - **Traduction du code** en langage d'assemblage vers un code objet

   Code produit : langage machine, plusieurs fichiers (1 par fichier source)
3. **Édition des liens**
   - Si le programme est constitué de plusieurs fichiers source et/ou si il utilise des fonctions et des variables externes (définies dans des bibliothèques extérieures), les bouts de code objet correspondants sont reliés entre eux pour former un **exécutable**.

   Code produit : langage machine, un exécutable

Compilation d'un fichier

Avec gcc (disponible sur les machines Un*x) :

```
$ gcc -o executable fichiersSource.c
```

Option -o = output = fichier produit

- Autres options pour n'effectuer que les étapes intermédiaires :
  - -E : pré-processeur
  - -S : génération du langage d'assemblage
  - -c : génération du code objet sans édition des liens

Options utiles :

- -Wall : affichage des avertissements (warnings)
- -O1, -O2, -O3 : optimisation du code

Exemple : compilation du fichier `helloWorld.c` pour produire l'exécutable `helloWorld` :

```
$ gcc -Wall -o helloWorld helloWorld.c
```

Premier programme C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5    printf( "Hello world\n" );
6    return EXIT_SUCCESS;
7 }
```

Point d'entrée dans le programme : la fonction `main`

- C'est une fonction : elle retourne un entier (type `int`)
  - Valeur de retour de la fonction `main` : valeur de sortie du programme (sous Un*x avec bash : `$?`)

Utilisation de fonctions extérieures : définition de ces fonctions

- Utilisation de `printf` : affichage formaté
- Fonction `printf` définie dans `stdio.h` : `#include <stdio.h>`

Utilisation d'une constante : `EXIT_SUCCESS` définie dans `stdlib.h`

- Inclusion du fichier où elle est définie : `#include <stdlib.h>`

Premier programme C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5    printf( "Hello world\n" );
6    return EXIT_SUCCESS;
7 }
```

Point d'entrée dans le programme : la fonction `main`

- C'est une fonction : elle retourne un entier (type `int`)
  - Valeur de retour de la fonction `main` : valeur de sortie du programme (sous Un*x avec bash : $?)

Utilisation de fonctions extérieures : définition de ces fonctions

- Utilisation de `printf` : affichage formaté
- Fonction `printf` définie dans `stdio.h` : `#include <stdio.h>`

Utilisation d'une constante : `EXIT_SUCCESS` définie dans `stdlib.h`

- Inclusion du fichier où elle est définie : `#include <stdlib.h>`

Premier programme C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5    printf( "Hello world\n" );
6    return EXIT_SUCCESS;
7 }
```

Point d'entrée dans le programme : la fonction `main`

- C'est une fonction : elle retourne un entier (type `int`)
  - Valeur de retour de la fonction `main` : valeur de sortie du programme (sous Un*x avec bash : $?)

Utilisation de fonctions extérieures : définition de ces fonctions

- Utilisation de `printf` : affichage formaté
- Fonction `printf` définie dans `stdio.h` : `#include <stdio.h>`

Utilisation d'une constante : `EXIT_SUCCESS` définie dans `stdlib.h`

- Inclusion du fichier où elle est définie : `#include <stdlib.h>`

Premier programme C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5    printf( "Hello world\n" );
6    return EXIT_SUCCESS;
7 }
```

Point d'entrée dans le programme : la fonction `main`

- C'est une fonction : elle retourne un entier (type `int`)
  - Valeur de retour de la fonction `main` : valeur de sortie du programme (sous Un*x avec bash : `$?`)

Utilisation de fonctions extérieures : définition de ces fonctions

- Utilisation de `printf` : affichage formaté
- Fonction `printf` définie dans `stdio.h` : `#include <stdio.h>`

Utilisation d'une constante : `EXIT_SUCCESS` définie dans `stdlib.h`

- Inclusion du fichier où elle est définie : `#include <stdlib.h>`

Déclaration de variables

Types de variables :

- Types entiers (integer) :
  - int : 4 octets, valeurs de -2^{31} à $2^{31}-1$
  - short int : 2 octets, de -32.768 à 32.767
  - long int : comme int
  - long long int : 8 octets, valeurs de -2^{63} à $2^{63}-1$
  - version non signées : pas de bit de signe
- Types caractères :
  - char : 1 octet, valeur ascii
  - unsigned char : caractère non signé
- Types réels :
  - float : nombre à virgule flottante simple précision (4 octets)
  - double : nombre à virgule flottante double précision (8 octets)
  - long double : 10 octets

Codage des nombres réels

Norme IEEE 754 : décomposition du nombre en trois éléments

- **bit de signe** si le nombre est signé
- **mantisse** : facteur multiplicateur retranché de 1 et compris entre 0 et 1
- **exposant**

En prenant s = signe, m = mantisse, e = exposant :

$$nombre = s \times (1 + m) \times 2^e$$

- Type `float` : 8 bits pour l'exposant et 23 bits pour la mantisse
- Type `double` : 11 bits pour l'exposant et 52 bits pour la mantisse
- Type `long double` : 15 bits pour l'exposant et 64 bits pour la mantisse

Déclaration et affectation de variables

Déclaration de variables :

`type nom;`

```
1   int hauteur;
2   float largeur, aire;
3   long double diagonale;
```

Affectation d'une valeur à une variable :

`nom = valeur;`

```
1   int hauteur;
2   float largeur;
3   hauteur = 42;
4   largeur = 0.59;
```

Toutes les lignes se terminent par un point-virgule (;)

Déclaration et affectation de variables

Déclaration de variables :

`type nom;`

```
1   int hauteur;
2   float largeur, aire;
3   long double diagonale;
```

Affectation d'une valeur à une variable :

`nom = valeur;`

```
1   int hauteur;
2   float largeur;
3   hauteur = 42;
4   largeur = 0.59;
```

Toutes les lignes se terminent par un point-virgule (;)

Déclaration et affectation de variables

Déclaration de variables :

`type nom;`

```
1        int hauteur;
2        float largeur, aire;
3        long double diagonale;
```

Affectation d'une valeur à une variable :

`nom = valeur;`

```
1        int hauteur;
2        float largeur;
3        hauteur = 42;
4        largeur = 0.59;
```

Toutes les lignes se terminent par un point-virgule (;)

Déclaration et affectation de variables

Déclaration de variables :

`type nom;`

```
1   int hauteur;
2   float largeur, aire;
3   long double diagonale;
```

Affectation d'une valeur à une variable :

`nom = valeur;`

```
1   int hauteur;
2   float largeur;
3   hauteur = 42;
4   largeur = 0.59;
```

Toutes les lignes se terminent par un point-virgule (;)

Transtypage (cast)

Possibilité de convertir une variable d'un type vers un autre type

- Attention à la compatibilité entre le type de départ et le type d'arrivée !

```
1  int hauteur;
2  float largeur;
3  hauteur = 42;
4  largeur = (float) hauteur;
```

Caractère vers entier :

- Utilisation des fonctions `atoi` (ascii to integer), `atol` (ascii to long)

Entier vers caractère :

- Écriture formatée de l'entier dans un caractère

Transtypage (cast)

Possibilité de convertir une variable d'un type vers un autre type

- Attention à la compatibilité entre le type de départ et le type d'arrivée !

```
1        int hauteur;
2        float largeur;
3        hauteur = 42;
4        largeur = (float) hauteur;
```

Caractère vers entier :

- Utilisation des fonctions `atoi` (ascii to integer), `atol` (ascii to long)

Entier vers caractère :

- Écriture formatée de l'entier dans un caractère