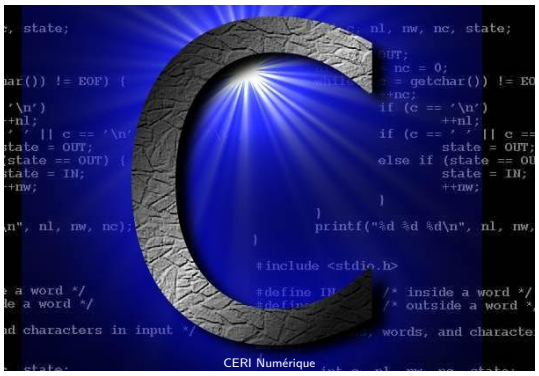
Algorithmique et langage C

— Premiers pas dans le monde merveilleux de la programmation —



Luc Fabresse
luc.fabresse@imt-nord-europe.fr

Organisation du cours

Algorithmique et Programmation avec le langage C

- 7 séances de 4h de Cours/TD/TP
- Évaluation : Qcm + TP notés

Citations

Je n'ai pas pour but d'éclairer ceux qui ne sont pas désireux d'apprendre, ni éveiller ceux qui ne sont pas soucieux de donner une explication eux-mêmes. Si je leur ai montré un angle du carré et qu'ils ne peuvent pas revenir à moi avec les trois autres, je ne devrais pas revenir sur le premier angle.
[Confucius]

Citations

Je n'ai pas pour but d'éclairer ceux qui ne sont pas désireux d'apprendre, ni éveiller ceux qui ne sont pas soucieux de donner une explication eux-mêmes. Si je leur ai montré un angle du carré et qu'ils ne peuvent pas revenir à moi avec les trois autres, je ne devrais pas revenir sur le premier angle.
[Confucius]

Règle d'or :

Soyez curieux !

Citations C

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.
[Waldi Ravens]

Writing in C or C++ is like running a chainsaw with all the safety guards removed
[Bob Gray]

Plan

Plan

Qu'est-ce que l'informatique ?

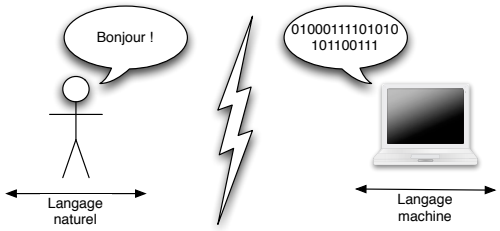
Objectif

Automatiser le traitement de données.

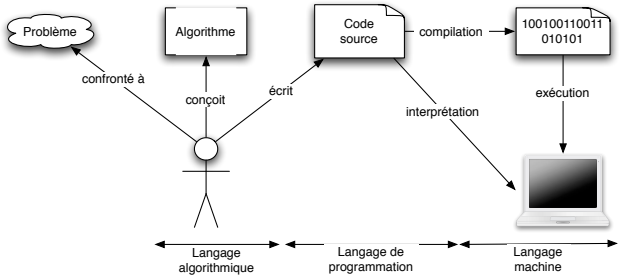
Comment ?

En utilisant des programmes (logiciels) afin qu'un (ou des) ordinateur(s) (matériels) effectuent le traitement désiré.

Problème



Principe général



Objectifs du cours

Objectif global

Savoir écrire des programmes en C permettant de résoudre des problèmes

Plus précisément :

- Savoir écrire des algorithmes
  - Définir et manipuler des structures de données complexes statiques et dynamiques
  - Posséder les notions de base sur la qualité et de la complexité d'un algorithme
- Savoir programmer en C
  - Connaître la syntaxe et la sémantique du C
  - Savoir utiliser quelques outils de programmation comme `gcc`, `Make`, ...

Qu'est-ce qu'un programme ?

Un programme comprend :

- Données
- Traitements

Aspect statique

Le *code source* est une description d'« objets informatique » (variables, constantes, ...) et d'« actions » (instructions, structures de contrôle, ...)

Aspect dynamique

Le comportement pendant l'exécution du programme c'est-à-dire lorsque l'ordinateur exécute la suite de changements d'états décrits dans le code

Langage de programmation

Objectif

Permettre l'écriture de programmes exécutables par un ordinateur afin d'automatiser une tâche

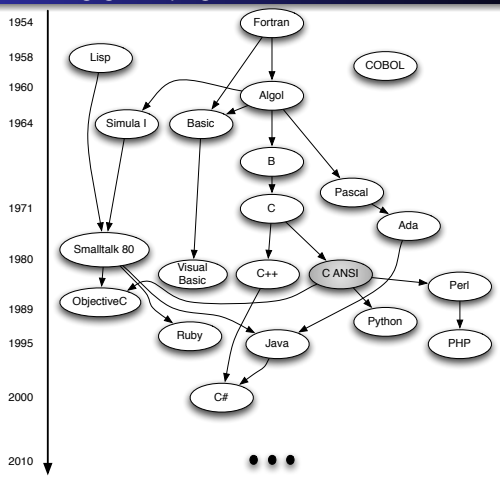
Caractéristiques

- Syntaxe
- Sémantique

De nombreuses familles (non disjointes) de langages de programmation :

- impératifs
- fonctionnels (Lisp, scheme, ...)
- déclaratifs (Oz)
- logiques (Prolog)
- à objets

Généalogie des langages de programmation



Le langage C

Principales caractéristiques

Évolué, structuré, bas niveau, peu typé

Quelques détails :

- différents types de données élémentaires : `int`, `char`, ...
- manipulation des données au niveau bits
- possibilité de définir de nouvelles structures de données plus évolués (tables, vecteurs, ...)
- accès direct à la mémoire (pointeurs)
- peu d'instruction (~40), aucune pour les entrées-sorties ni les structures de données complexes
- structures de contrôle classiques : itérations, sélections, sous-programmes
- composition et imbrication de structures de contrôle
- récursivité
- beaucoup de bibliothèques
- compilation séparée

Exemple de programme C : l'indispensable « Hello World »

Code source

```c
/* Mon programme Hello World */
#include <stdio.h>

int main(void) {
  printf("Hello World !\n"); // Affichage sur la sortie standard
  return 0;
}
```

Compilation et exécution

```
$ gcc -Wall -o hello_world hello_world.c
$ ./hello_world
```

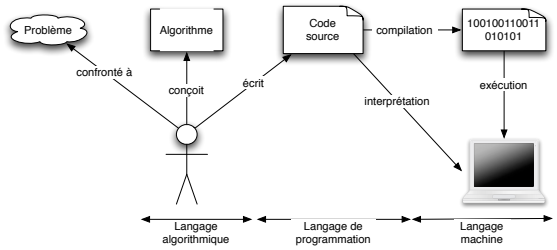
Résultat de l'exécution

```
1 Hello World !
```

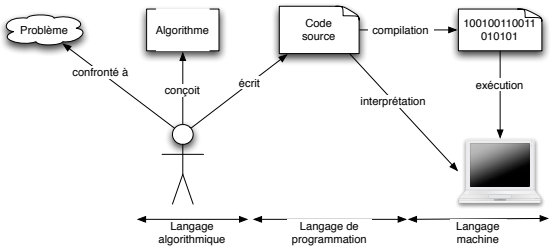
Testez les exemples simplement via une page Web

`https://www.tutorialspoint.com/compile_c_online.php`
`https://repl.it`

Pourquoi écrire des algorithmes ?



Pourquoi écrire des algorithmes ?



Utiliser le langage algorithmique permet de :

- traduire une solution d'un problème en une solution informatique
- ne pas mélanger étapes de solution et détails d'un langage de programmation
- se concentrer sur les éventuelles erreurs sémantiques et non syntaxiques
- proposer une solution générique qui ne dépend pas d'un langage de programmation et donc traduisible vers plusieurs langages de programmation

Luc Fabresse – Cours Algo&C 16

Littérature

Brian-W Kernighan, Dennis-M Ritchie
Le langage C : Norme ANSI, 2e édition, Dunod, 2004.

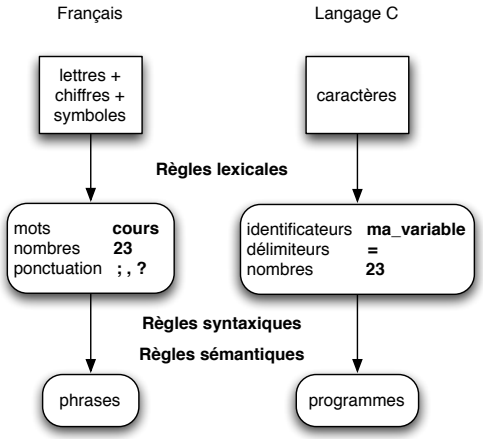
Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein
Introduction à l'algorithmique, 2e édition, Dunod, 2004.

Gnu C Library Documentation
`http://www.gnu.org/software/libc/manual/`

Plan

Luc Fabresse – Cours Algo&C 18

Analogie langage naturel/langage de programmation



Plan

Luc Fabresse – Cours Algo&C 20

Éléments lexicaux (1)

Le jeu de caractère de base

- lettres minuscules, majuscules et chiffres : a..z ∪ A..Z ∪ 0..9
- caractères de soulignement : _
- caractères non visibles : espace, tabulation, retour arrière, nouvelle ligne, saut de page
- caractères spéciaux et de ponctuation :
  `, . ; : ' " ( ) [ ] { } < ! | / \ ~ + # % & ^ * - = >`

Mots réservés

auto break case char const continue default do double else enum extern float for goto if inline (C99) int long register restrict (C99) return short signed sizeof static struct switch typedef union unsigned void volatile while _Bool (C99) _Complex (C99) _Imaginary (C99)

Commentaires

- multi-lignes : `/* ... */`
- en ligne : `// ...`

Éléments lexicaux (2)

Indentificateurs

- Utilisés pour nommer les « éléments » du programme : variables, constantes, fonctions, ...
- Règles de nommage :
  - 1er caractère : A..Z,a..z,_
  - Caractères suivants : A..Z,a..z,0..9,_

Exemples : _J, fin_de_fichier, x33

Règles et conventions

- Une instruction sur une ou plusieurs lignes
- Plusieurs instructions sur une seule ligne
- La fin d'une instruction est toujours marquée par le caractère ;
- Les mots réservés et fonctions prédéfinies sont en minuscules
- Quelques constantes sont en majuscules (`NULL`,`EOF`,...)

Luc Fabresse – Cours Algo&C 22

Plan

Types de base du C

Types historiques

Type	Mot-clé	Valeurs	Taille
Types de données entières			
char		$[-2^7..2^7-1]$	8 bits / 1 octet
unsigned char		$[0..2^8-1]$	8 bits / 1 octet
int/short		$[-2^{15}-1..2^{15}-1]$	16 bits / 2 octets
unsigned int/short		$[0..2^{16}-1]$	16 bits / 2 octets
long		$[-2^{31}-1..2^{31}-1]$	32 bits / 4 octets
unsigned long		$[0..2^{32}-1]$	32 bits / 4 octets
long long (C99)		$[-2^{63}-1..2^{63}-1]$	64 bits / 8 octets
unsigned long long (C99)		$[0..2^{64}-1]$	64 bits / 8 octets
Types de données réelles			
Réel simple précision	float	valeur absolue entre $3.4*10^{-38}$ et $3.4*10^{38}$	32 bits / 4 octets
Réel double précision	double	valeur absolue entre $1.7*10^{-308}$ et $1.7*10^{308}$	64 bits / 8 octets
Réel long double précision	long double	valeur absolue entre $3.4*10^{-4932}$ et $3.4*10^{4932}$	80 bits / 10 octets

Types entiers C99

http://en.cppreference.com/w/c/types/integer

Fixed width integer types (since C99)

Types

Defined in header <stdint.h>

int8_t **int16_t** **int32_t** **int64_t**	signed integer type with width of exactly 8, 16, 32 and 64 bits respectively with no padding bits and using 2's complement for negative values (provided only if the implementation directly supports the type)
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	fastest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
int_least8_t **int_least16_t** **int_least32_t** **int_least64_t**	smallest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
intmax_t	maximum width integer type
intptr_t	integer type capable of holding a pointer
uint8_t **uint16_t** **uint32_t** **uint64_t**	unsigned integer type with width of exactly 8, 16, 32 and 64 bits respectively (provided only if the implementation directly supports the type)
uint_fast8_t **uint_fast16_t** **uint_fast32_t** **uint_fast64_t**	fastest unsigned integer type with width of at least 8, 16, 32 and 64 bits respectively
uint_least8_t **uint_least16_t** **uint_least32_t** **uint_least64_t**	smallest unsigned integer type with width of at least 8, 16, 32 and 64 bits respectively
uintmax_t	maximum width unsigned integer type
uintptr_t	unsigned integer type capable of holding a pointer

Autres Types de base

Absence de type

`void`

Pas de type booléen

Simulation par les int :

- Faux représenté par l'entier 0
- Vrai par n'importe quel entier sauf 0

`stdbool.h` (C99)

- type `bool`
- littéraux : `true` (1) et `false` (0)

Attention, ces macros peuvent être redéfinie...

Autres Types de base

Absence de type

`void`

Pas de type booléen

Simulation par les int :

- Faux représenté par l'entier 0
- Vrai par n'importe quel entier sauf 0

`stdbool.h` (C99)

- type `bool`
- littéraux : `true` (1) et `false` (0)

Attention, ces macros peuvent être redéfinie...

Pas de type chaîne de caractère

La chaîne `"Bonjour"` considérée comme des tableaux de caractères

Plan

L'instruction `typedef`

Syntaxe `typedef <type existant> <nouveau nom>`

Effet déclaration du type `<nouveau nom>`

Exemple

```c
#include <stdlib.h>
#include <stdint.h>

typedef int32_t MesEntiers;

MesEntiers secretCalculus() {...}

int main(void) {
  MesEntiers resutlat = 0;
  resutlat = secretCalculus();

   return EXIT_SUCCESS;
}
```


Plan

Variable

Variable = ⟨*Identificateur*, *Type*, *Valeur*⟩

- Un type C décrit la représentation mémoire
- Une variable doit **toujours** être déclarée avant d'être utilisée

Exemples

```
float x;  // déclaration d'une variable réelle x
float y, z;  // déclaration de deux variables réelles y et z
int i = 0;  // déclaration d'une variable entière i initialisée à 0
int j = 1, k = 0;
char un_caractere = 'C';
```

Constantes

Constantes nommées

```
const float PI = 3.14159;
```

Littéraux

- Entier décimal : 789
- Entier hexadécimal : 0xfe45
- Caractère :
  - imprimables : 'a'
  - non imprimables : '\n' (saut de ligne), '\t' (tabulation), '\b' (backspace) '\r' (retour chariot), '\\' (basckslach), '\'' (apostrophe), '\"' (guillement)
- Réel : 1.35, 2.345e-2, 6.02e23
- Chaîne : "ceci est une chaine\n"

Opérateurs arithmétiques

Binaires

- + : addition
- - : soustraction
- * : multiplication
- / : division
- % : modulo *i.e* reste de la division entière

Unaires

- - : change le signe d'un nombre positif à négatif et vice-versa
- ++ : pré/post-incrémentation
- - : pré/post-décrémentation

Exemples

```
int i=2,j,k;
j = ++i; // ici j vaut 3 et i vaut 3
k = --j; // ici k vaut 2 et j vaut 2
```

```
int i=2,j,k;
j = i++; // ici j vaut 2 et i vaut 3
k = j--; // ici k vaut 2 et j vaut 1
```


Opérateurs logiques

! (non logique)

Exemple : !b

&& (ET logique) : opérateur binaire **paresseux**

Dans l'expression p && q && r, r n'est pas évalué si :

- p est faux
- ou bien si p est vrai et q est faux

|| (OU logique) : opérateur binaire **paresseux**

Dans l'expression p || q || r, r n'est pas évalué si :

- p est vrai
- ou bien si p est faux et q est vrai

Opérateurs de comparaison

Opérateurs

- < : inférieur
- <= : inférieur ou égal
- > : supérieur
- >= : supérieur ou égal
- == : égal (test d'égalité !)
- != : différent

Opérandes

- Nombres
- Pointeurs

Résulat

- 1 si l'expression est vraie
- 0 sinon

Opérateurs de manipulation de bits

Unaire

~ : complément à 1 bit à bit

Binaires

Dont les opérandes sont des int ou short :

- << : décalage à gauche
- >> : décalage à droite
- & : ET bit à bit
- | : OU bit à bit
- ^ : OU EXCLUSIF bit à bit

Affectation

Affectation simple

Syntaxe <variable> = <expression>

Effet
1. la valeur *v* de expression est évaluée
2. la valeur de variable est fixée à *v*
3. une affectation est une **expression** dont la valeur est celle affectée (*v*)

Exemples

```
int a, b, c = 3;
b = 5;
a = b = c; // a,b et c valent 3
```

Affectation combinée d'un opérateur

Syntaxe <variable> <operateur> = <expression>

- <operateur> : + - * / % << >> & ^ |

Exemple

```
i += j; // équivalent à i = i + j
```


Autre opérateurs (1)

Opérateur unaire cast pour les conversions de type explicite

Syntaxe (<type>) <expression>

Effet Converti le type de l'expression vers le type choisi. Attention aux pertes de données !

Exemples

```
int pi = (int) 3.14;

char c;
int i = 65;
c = (char) i; // c vaut 'A' dont le code ASCII est 65
```

Opérateur ternaire pour les conditions

Syntaxe <condition> ? <expr1> : <expr2>

Effet La valeur de cette expression est celle de expr1 si la valeur de l'expression condition est différente de 0, celle de expr2 sinon

Exemple i = (j>10)? (j/2) : (j*2);

Autre opérateurs (2)

Taille en mémoire : sizeof

Syntaxe `sizeof(<type>)` ou `sizeof <expression>`

Effet Rend la taille en octets de son opérande

Exemple

```
int i,j;
i = sizeof(float);
j = sizeof(i);
```

Dans la suite (cf. Pointeurs)

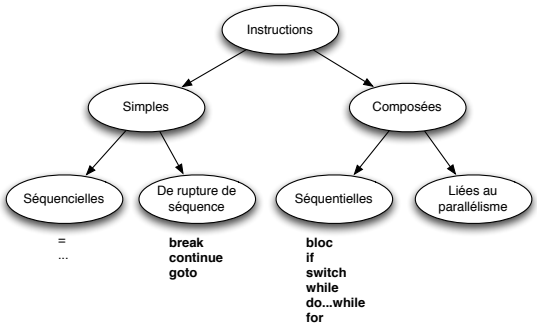
- Opérateurs d'indirection (*,->)
- Opérateur d'adressage (&)

Priorité des opérateurs

Catégorie d'opérateur	Opérateur
Divers	`() [] . ->`
Unaires	`! ~ - ++ -- & * (type) sizeof`
Arithmétiques	`* / % + -`
De décalage de bits	`<< >>`
De comparaison	`< <= > >= == !=`
Binaires de bits	`& ^ |`
Logiques	`&& || condition?then:else`
D'affectation	`= *= /= %= += -= ^= &= <<= >>=`
De séquence	`,`

Plan

Vue globale des instructions en C



Bloc d'instructions

Syntaxe

```
{  // début du bloc
    <liste de déclaration de variables>
    <liste d'instructions>
}  // fin du bloc
```

Exemple

```
{
  int i, somme = 0, resultat;
  for ( i=0; i<10; i++ )
    somme += f(i) * 2;
  resultat = somme < 100;
}
```

Remarques

- Possibilité d'imbrication des blocs
- Variables locales à un bloc
- Attention aux homonymies ! (cf. *portée, visibilité et masquage*)

If

Syntaxe

```
if( <expression> )
    <instruction1>
else                  // optionnel
    <instruction2>    // optionnel
```

Exemple

```
if(a>b)
  max = a;
else {
  max = b;
  if(a==b) egal = 1;
}
```

Sémantique

- Si l'évaluation de `<expression>` est différente de 0 `<instruction1>` est exécutée,
- sinon `<instruction2>` est exécutée (si elle existe)

Switch

Syntaxe

```
switch( <expr_discriminante> ){
    case <expr_constante1>:
        <instruction1>
    case <expr_constante2>:
        <instruction2>
    ...
    default:
        <instructionN>
}
```

Exemple

```
switch(commande){
  case 'a':
    printf("Ajout"); ...;
    break;
  case 'm':
    printf("Modif"); ...;
    break;
  default:
    printf("Commande inconnue");
}
```

Sémantique

- La valeur *v* de `<expr_discriminante>` doit être de type `int`, `char` ou `enum`
- *v* est comparée aux `<expr_constanteI>`
- Dès lors qu'il y a égalité, les instructions sont exécutées à partir d'`<instructionI>` et jusqu'à la rencontre d'une instruction `break` ou la fin du `switch`
- Si aucune `<expr_constanteI>` n'est égale à *v*, l'instruction de la clause `default` est exécutée (si elle existe)

While (tant que)

Syntaxe

```
while( <expr> )
    <instruction>
```

Sémantique

- Au début de chaque itération, `<expr>` est évaluée
- Si sa valeur est différente de 0, `<instruction>` est exécutée puis la boucle recommence
- `<instruction>` peut ne jamais être exécutée !

Exemple

```
int i = 0, resultat = 0;
while( i<=10 ){
  resultat += i;
  i++;
}
```

Do ... While (faire ... tant que)

Syntaxe

```
do <instruction> while (<expression>);
```

Sémantique

- `<instruction>` est exécuté au début de chaque itération
- A la fin de chaque itération, `<expression>` est évaluée
- Si sa valeur est différente de 0, la boucle recommence
- `<instruction>` est exécuté au moins 1 fois !

Exemple

```
int i = 0, resultat = 0;
do {
  resultat += i;
  i++;
} while( i<=10 );
```


For (pour)

Syntaxe

```
for(<expression1>;<expression2>;<expression3>)
    <instruction>
```

- `<expression1>` : initialisation des variables de contrôles
- `<expression2>` : condition d'itération de la boucle
- `<expression3>` : actualisation des variables de contrôles
- N'importe laquelle de ces 3 expressions peut être omise
- Quand `<expression2>` est omise : boucle infinie
- L'opérateur , peut être utilisé dans `<expression1>` et `<expression3>` pour spécifier plusieurs expressions

```
for( i=0,j=f(); i<j; i++,j--) { ... }
```

Sémantique équivalente à :

```
<expression1>;
while(<expression2>) {
    <instruction>
    <expression3>
}
```

Exemple

```
int i, resultat = 1;
int exposant = 3, nombre = 5;
for (i = 0; i < exposant; i++)
  resultat *= nombre;
```


Instructions de rupture de séquence (1)

break

Arrêt de la 1re instruction répétitive englobante

```
int i, j;
for (i = 0; i < 100; i++) {
    j = f(i);
    if (j < 0) break; /*arrêt de la boucle si j négatif*/
    printf("Valeur f(%d) = %f",i,j);
}
```

continue

Provoque l'arrêt de l'itération courante et le passage au début de l'itération suivante

```
int i;
for(i=1;i<=10;i++) {
  if(i%2) continue;
  printf("%i\n",i);
}
```

goto

Permet de se brancher à un emplacement donné du programme dans la même fonction

```
for(i=1; i<=10; i++) {
  if (i==2) goto fin;
  printf("iteration %d\n", i+1);
}
fin : printf("->fin\n");
```


Instructions de rupture de séquence (2)

À savoir :

- Rend difficile la compréhension des programmes
- Grande source d'erreur en cas de modification du programme
- Très peu utile si bonne conception (sauf `break` dans les `swicth` par exemple)

Luc Fabresse – Cours Algo&C 49

Instructions de rupture de séquence (2)

À savoir :

- Rend difficile la compréhension des programmes
- Grande source d'erreur en cas de modification du programme
- Très peu utile si bonne conception (sauf `break` dans les `swicth` par exemple)

Règles d'or :

- Limiter l'usage de `break` et `continue`
- Ne jamais utiliser de `goto`

http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html

Plan

Fonctions

Qu'est-ce qu'une fonction ?

Un traitement ayant des entrées (paramètres) et une sortie (résultat)

Caractéristiques

- Passage de paramètres par valeur uniquement
- Simulation du passage par référence avec les pointeurs
- Pas d'imbrication des définitions des fonctions
- Récursivité autorisée
- Un seul point d'entré : 1re instruction du corps de la fonction
- Plusieurs points de sortie *possibles* à l'aide de l'instruction `return`

Définition d'une fonction

Syntaxe

```
<type resultat> <nom fonction> (<declaration des paramètres>)
    <instruction>
```

Type résultat

- `void` si aucun résultat retourné
- sinon préciser, exemple : `int`
- types autorisés :
  - type de base, pointeur, structure
  - pas de tableaux !

Définition d'une fonction (2)

Exemple

```c
float puissance(float f,int exp){
  float resultat=1;
  int i;   // declaration de variables locales
  for(i=exp;i>0;i--)
    resultat *= f;
  return resultat;
}
```


La fonction `main` : LE point d'entrée du programme

`main` : les valeurs de retour

```c
#include <stdlib.h>

int main(void){
  // ...

  //return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

`main` : les paramètres

```c
#include <stdlib.h>

int main(int argc, char* argv[]){
  // ...

  //return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```


Appel d'une fonction (1)

L'appel d'une fonction est une expression

Syntaxe

`<nom de la fonction>(<liste de paramètres effectifs>)`

Exemple

```c
float f = 2.0;
int i;
for(i=0; i<10; i++)
  printf("2^%d = %f",i,puissance(f,i));
```

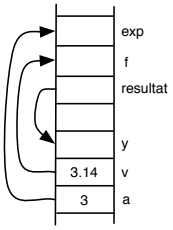

Passage de paramètres par valeur

Fonctionnement

Copie des valeurs des paramètres réels dans les paramètres formels lors de l'appel

```c
float puissance(float f,int exp){
  float resultat=1;
  // ...
  return resultat;
}

int main(void){
  float v = 3.14;
  int a = 3;
  float y = puissance(v,a);
}
```



Conséquences

- La fonction manipule des copies
- Modifier les copies est sans incidence sur les paramètres réels !

Plan

Récursivité directe

Définition

Un algorithme est dit récursif directement quand il fait appel à lui même dans sa définition. Exemple : A appelle A

Récursivité indirecte (croisée)

Un algorithme est dit récursif indirectement s'il fait appel à lui même de manière indirecte. Exemple : A appelle B et B appelle A.

Récursivité linéaire

Une récursivité est dite linéaire lorsqu'un algorithme fait un seul appel à lui même qu'il soit direct ou indirect

Exemple de récursivité linéaire : la fonction `puissance`

Analyse du problème

- Les paramètres : un nombre X et la valeur de la puissance N
- Le cas trivial : $X^0 = 1$
- Le cas général : $X^n = X * X^{n-1}$

La fonction `puissance_r`

```c
float puissance_r(float f,unsigned int exp){
  if(exp==0)
    return 1.0;

  return f*puissance_r(f,exp-1);
}
```

Récursivité terminale

Une fonction à récursivité terminale est une fonction où l'appel récursif est la dernière instruction à être évaluée.

Exemple de récursivité terminale

La fonction `puissance_recursive_terminale`

```c
float puissance_recursive_terminale_acc(float f,unsigned int exp, float acc){
  if(exp==0)
    return acc;
  return puissance_recursive_terminale_acc(f,exp-1,f*acc);
}

float puissance_recursive_terminale(float f,unsigned int exp){
  return puissance_recursive_terminale_acc(f,exp,1);
}
```

Récursivité terminale

Optimisation du compilateur qui transforme le binaire en itération

Exemple de récursivité non linéaire : Suite de Fibonacci

Analyse du problème

- Les paramètres : un nombre N
- Le cas général : $u_n = u_{n-1} + u_{n-2}$
- Les cas triviaux : $u_1 = 1$ et $u_2 = 2$

fibonacci(N : Entier) : Entier

```
begin
    if N = 1 then
        return 1;
    else
        if N = 2 then
            return 2;
        else
            return fibonacci(N-1) + fibonacci(N-2);
        end
    end
end
```


Plan

Notion de « portée »

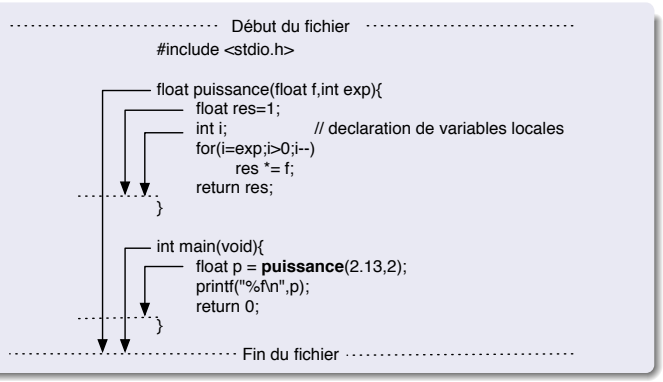
La portée d'une déclaration (type, constante, variable, fonction) :

- est la région du programme dans laquelle cette déclaration a un effet
- commence juste après la déclaration et s'étend jusqu'à la fin du corps englobant

Conséquences

- Une entité n'est pas accessible (et ne peut donc pas être utilisée) tant qu'elle n'a pas été déclarée
- Une entité déclarée avant un sous-programme (bloc,fonction) est utilisable dans ce sous-programme
- La portée d'une entité déclarée dans un sous-programme (paramètre formel,variable locale) s'étend à ceux déclarés à l'intérieur après elle

Notion de « portée » : exemple



Notion de « portée »

Problème

```
int main(void){
  double x,y;
  int i;
  for( i=0; i<90; i++ ){
    y=(double) i;
    x=sin(y); // sin inconnue ici !
    printf("sin(%d)=%f\n",i,x);
  }
}

double sin(double angle){
  // ...
}
```

Appel de la fonction sin alors qu'elle n'est pas encore définie !

Solution 1

```
double sin(double angle){
  // ...
}

int main(void){
  // ...
}
```


Notion de « prototype » (ou « signature ») de fonction

Problème

Comment définir deux fonctions qui s'appellent mutuellement ?

Solution

- Déclarer le prototype de l'une d'abord
- Syntaxe : `<type> <nom de la fonction>(<type> <nom parametre>, ...)`
- Le nom des paramètres formels est facultatif lors de la déclaration d'un prototype

Exemple

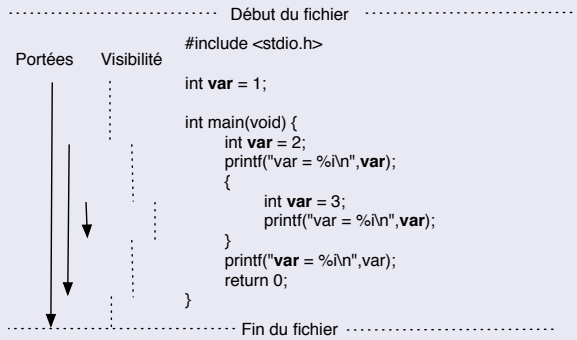
```
double f(double); // déclaration du prototype (signature) de f

int g(double b){
  f(i); // f n'est pas définie mais a bien été déclarée donc à portée et est visible
}

double f(double a) {
  b=g(a-1); // ...
}
```


Notion de « visibilité » et de « masquage »

Cas d'homonymie d'identificateurs



Effets de bord (1)

Un effet de bord est :

une modification de l'environnement d'appel indépendamment du mécanisme de passage des paramètres

Exemple typique

```
float resultat;

int g(double b){
  //...
  resultat = i; // modification d'une variable globale
  //...
}

int main(void){
  // ...
  g(j);
  // ...
  ok = resultat > 10;
  // ...
}
```

Effets de bord (2)

À savoir :

- Les effets de bord ne sont utiles que dans des cas rares
- Ils rendent la compréhension des programmes difficile
- Ils introduisent des dépendances liées à l'ordre d'exécution

Effets de bord (2)

À savoir :

- Les effets de bord ne sont utiles que dans des cas rares
- Ils rendent la compréhension des programmes difficile
- Ils introduisent des dépendances liées à l'ordre d'exécution

Règle d'or :

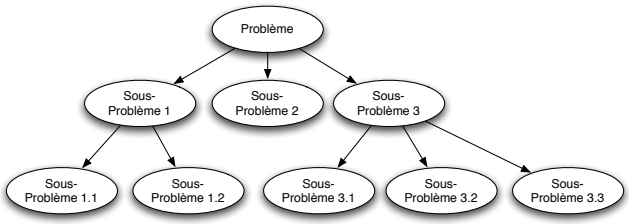
Ne jamais utiliser d'effets de bord !

Plan

Conception descendante : principe général

Principe

« *Divide ut regnes* » [Machiavel,1532]



Conception descendante et C

Utilisation des fonctions

- Une fonction pour chaque sous-problème
- Deux sous-problèmes identiques utilisent la **même** fonction (réutilisation)

Pour chaque fonction, identifier :

- Les données en entrées (les paramètres)
- Les résultats en sortie (valeurs de retour)
- Les entrées/sorties (écran, fichier, ...)
- Les erreurs possibles (valeur d'un paramètre en dehors d'un intervalle, ...)

Analyse récursive

Principe

Pour trouver une solution récursive à un problème, on cherche à le décomposer en plusieurs sous-problèmes de même type mais de taille inférieure. Méthode pour écrire un algorithme récursif :

- Paramétrage du problème : on détermine les éléments dont dépend la solution et qui caractérisent la taille du problème
- Recherche d'un cas trivial et de sa solution. C'est la clause de finitude qui permet de faire l'évaluation sans récursivité et donc d'éviter la récursivité à l'infini
- Décomposition du cas général en cas plus simples, eux-mêmes décomposables pour aboutir au cas trivial

Plan

Écriture de programmes C

Commencer par écrire les prototypes des fonctions :

- dans des fichiers *header* (*.h*)
- 1 fichier *header* par groupe logique de fonctions. Exemple :
  - `math.h` (fonctions mathématiques)
  - `stdio.h` (fonctions d'entrées/sorties standard)
  - `sudoku.h`

Puis, écrire les définitions des fonctions :

- dans des fichiers sources (*.c*)
- inclure les fichiers *.h* adéquats (`#include "sudoku.h"`)

Structure d'un fichier *header*

Structure d'un fichier *header* (*.h*)

```c
/*** directives de compilation ***/

/*** appels aux bibliothèques ***/
#include <stdio.h>

/*** déclaration de constantes, types, variables globales ***/
const float TVA = 0.196;

/*** déclaration de signatures de fonction ***/

/* retourne la factorielle de l'entier passé passé en paramètre */
unsigned int factorielle(unsigned int); // signature de fonction
```

Structure d'un fichier source

Structure d'un fichier source (*.c*)

```c
/*** directives de compilation ***/

/*** appels aux bibliothèques ***/
#include <stdio.h>

/*** définitions de fonction ***/

int g(double b) // signature de fonction
{ // corps de fonction
  //...
}

int main(void){ // fonction principale, point d'entré du programme
  // ...
  return 0;
}
```

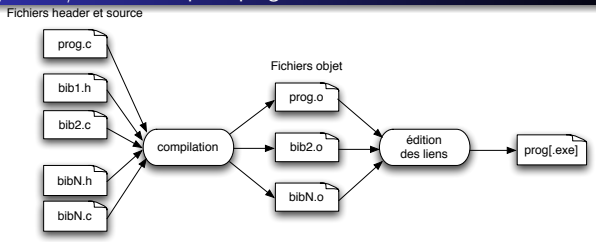

Compilation/exécution : cas des programmes simples (1 fichier source)

Commande :

`gcc -W -Wall -std=c99 -o prog prog.c`

- `<fichier source>` : nom du fichier source à compiler
- Options :
  - `-W` et `-Wall` : `Warning all` *i.e* affiche de nombreuses mises en gardes lors de la compilation
  - `-std=c99` : utilise la norme 1999 du C
  - `-o <nom exécutable>` : nom du fichier exécutable à générer (par défaut `a.out` si cette option est omise)

Compilation/exécution : principe général



Commande :

```
gcc -W -Wall -std=c99 -c bib1.c
gcc -W -Wall -std=c99 -c bib2.c
gcc -W -Wall -std=c99 -c bib3.c
gcc -W -Wall -std=c99 -c bibN.c
gcc -W -Wall -std=c99 -o prog prog.c bib1.o bib2.o bib3.o bibN.o
                              OU
gcc -W -Wall -std=c99 -o prog prog.c bib1.c bib2.c bib3.c bibN.c
```


Erreurs et avertissements

Exemple

```
$ gcc -Wall -o puissance puissance.c
puissance.c: In function 'puissance':
puissance.c:6: error: syntax error before 'int'
puissance.c:7: error: 'i' undeclared (first use in this function)
puissance.c:7: error: (Each undeclared identifier is reported only once
puissance.c:7: error: for each function it appears in.)
puissance.c: At top level:
puissance.c:12: warning: return type of 'main' is not 'int'
```

Les causes les plus courantes

- Manque de point-virgule
- Manque parenthèse ou accolade fermante
- Variable ou fonction non déclarée
- Variable non initialisée
- Utilisation de = au lieu de ==
- Bibliothèque non importée
- ...

Plan

Généralités sur les entrées/sorties

Pré-requis

Inclure la bibliothèque d'E/S standard de C (`stdio.h`) via la commande d'inclusion du pré-processeur (`#include`)

Exemple

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

La bibliothèque `stdio.h`

- Entrée : saisie utilisateur au clavier
- Sortie : affichage textuel à l'écran

Entrées/Sorties de caractères

`int getchar(void)`

- Rend le prochain caractère de `stdin` sous forme d'entier
- Quand la fin de fichier est détectée, rend la valeur particulière `EOF`

`void putchar(int)`

Met le caractère passé en argument sur `stdin`

Exemple

```c
#include <stdio.h>

int main(void){
  int c;
  while ((c=getchar()) != EOF)
    putchar(c);
  return 0;
}
```

printf : sorties formatées

Effet de la fonction printf

Affiche des données de type de base sur la sortie standard (stdin, par défaut l'écran)

Particularité de printf

Admet un nombre variable d'arguments

int printf(<format>, <arg1>, <arg2>, ... <argn>);

- <format> est une chaîne contenant 2 types d'objets :
  - des caractères à copier tels quels
  - des spécificateurs de format (commençant par %) qui définissent comment afficher les arguments <arg i>
- printf retourne le nombre de caractères affichés. Permet de détecter les erreurs.

Exemple d'utilisation de printf

Code

```
int i = 3;
float r = 5.1234567;
printf("Variables :\ni = %+d \nr = %8.3f\n", i, r);
```

Résultat à l'écran

```
Variables :
i = +3
r =   5.123
```


Spécification de format

Syntaxe : %[+][<nombre1>][.<nombre2>]<carac>

- + : ajoute le signe pour les nombres
- <nombre1> : taille minimum du champ d'impression
- <nombre2>
  - pour une chaîne : nombre max de caractères
  - pour un réel : nombre de chiffres après la virgule
- <carac> est le format de la donnée

Formats de données

Format	Signification
%c	Caractère ASCII (char)
%s	Chaîne de caractères
%d	Entier décimal signé (int)
%i	Nombre entier (int) lu en base 16 s'il commence par 0x ou 0X ; lu en base 8 s'il commence par un 0 ; dans tous les autres cas, lu en base 10
%o	Nombre entier octal
%u	Nombre entier non signé (unsigned int)
%x %X	Entier hexadécimal signé (unsigned int)
%f	Nombre à virgule flottante
%e %E	Nombre à virgule flottante en format exponentiel
%g %G	Nombre à virgule flottante aux formats %f ou %e
%p	Pointeur
%n	Pointeur (nombre de caractères déjà donnés)

On peut ajouter un *modificateur* de taille entre le % et la lettre de format (uniquement pour les formats dioux) :

Modificateur	Signification
%hh	donnée de taille char
%h	donnée de taille short
%l	donnée de taille long
%ll	donnée de taille long long

Séquences d'échappement

Une séquence d'échappement est :

- une suite de caractères commençant par un \
- considérée comme 1 seul caractère (de type char)

Séquence	Signification
\n	saut de ligne
\t	tabulation horizontale
\v	tabulation vericale
\b	effacer un caractère (backspace)
\r	retour chariot
\f	saut de page
\a	émet un bip sur le haut parleur interne
\'	affiche une apostrophe
\\	affiche le caractère backslash
\"	affiche un guillement
\ddd	affiche des codes ASCII en notation octale
\xddd	affiche des codes ASCII en notation hexadécimale

Exemple

```
printf("\033[2J"); // efface l'écran
```


scanf : entrées formatées

Fonction scanf

Lit les caractères dans le buffer d'entrée standard, les interprète et range les résultats aux adresses mémoire fournies. Cette fonction admet une nombre variable d'arguments.

scanf(char *format, ...)

- format est une chaîne de caractères dirigeant l'interprétation des caractères lus (cf. cours2). La saisie s'arrête lorsque la chaîne de format est consommée ou qu'elle ne correspond pas à la saisie
- les paramètres supplémentaires sont généralement des noms de variables précédés du symbole & (cf. pointeurs)
- retourne le nombre de saisies correctement effectuées
  - 0 indique qu'aucun élément n'a pû être affecté
  - EOF indique la fin de fichier
- Lorsqu'un appel à scanf échoue (le caractère saisi ne correspond pas à celui de la chaîne de format), le prochain appel à scanf reprends immédiatement après le dernier caractère déjà converti

Résultat de scanf sans *erreur*

Toutes les variables ont été correctement affectées

Exemple d'utilisation de scanf

Code

```
int i;
float f;
scanf("%d %f", &i, &f);
printf("i = %d f = %f\n", i, f);
```

Résultat à l'écran

```
23
34.9
i = 23 f = 34.900002
```


Exemple scanf avec redirection d'entrée

```
#include <stdio.h>
int main(void) {
   double sum, v;
   sum = 0;
   while (scanf("%lf", &v) == 1)
      printf("\t%.5f\n", sum += v);
   return 0;
}
```

```
$ gcc -Wall readFile.c -o readFile
$ ./readFile < data.txt
./readFile < data.txt
   12.30000
   31413413.13413
   0.13413
   1.13410
   13413.13400
```

```
$ cat data.txt
  12.3
  31413413.1341341
  0.134134
  1.1341 13413.134
```


Redirection d'Entrée/Sortie

Code

```
#include <stdio.h>

int main(void) {
  int i, sum = 0, nb = 0;
  printf("Entrer un nombre: ");
  scanf("%d",&i);
  while( i != 0 ) {
    nb++;
    sum += i;
    printf("Entrer un nombre: ");
    scanf("%d",&i);
  }
  printf("\nmoyenne = %.2f\n",
    (float)sum/nb);
  return 0;
}
```

Redirection de sortie

```
$ ./mean > resultat.txt
2
3
0
$ cat resultat.txt
Entrer un nombre: Entrer un nombre: Entrer un nombre:
moyenne = 2.50
```

Redirection d'entrée et de sortie

```
$ ./mean < donnees.txt > resultat.txt
$ cat resultat.txt
Entrer un nombre: Entrer un nombre: Entrer un nombre:
moyenne = 2.75
```

À ne pas faire : `prog.c`

```c
// expliquez ce que fait ce code ?
// oui, il fait quelque chose, testez-le
#include <stdio.h>
int main(void){int k=0;float i,j,r,x,y=-16;while(puts(""),y++<15)
for(x=0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))
for(i=k=r=0;j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);return 0;}
```

Règles

- *indenter* !
- nommer clairement les fichiers, variables, fonctions, ...
- commenter !