Compléments de programmation et d'algorithmique

Cours 1:
Introduction
Langage C

J. Roland

QuIC

Quantum Information & Communication

Aujourd'hui

Plan

L'Université Virtuelle

Conseil

Inscrivez-vous dès à présent sur l'UV !

- Si inscription au BA3 en ordre : automatique
- Sinon : possibilité d'inscription manuelle
- Pas d'accès à l'UV ? Voir page web du service QuIC

Répondez aux questions suivantes :
Cours à distance ou en présentiel
Option Informatique ou Biomédical

Pourquoi l'UV ?

- Slides des cours théoriques
- Enoncés et corrigés des TPs
- Code de base pour cours théoriques et TPs
- Nombreux liens vers ressources en ligne
- Instructions pour la machine virtuelle

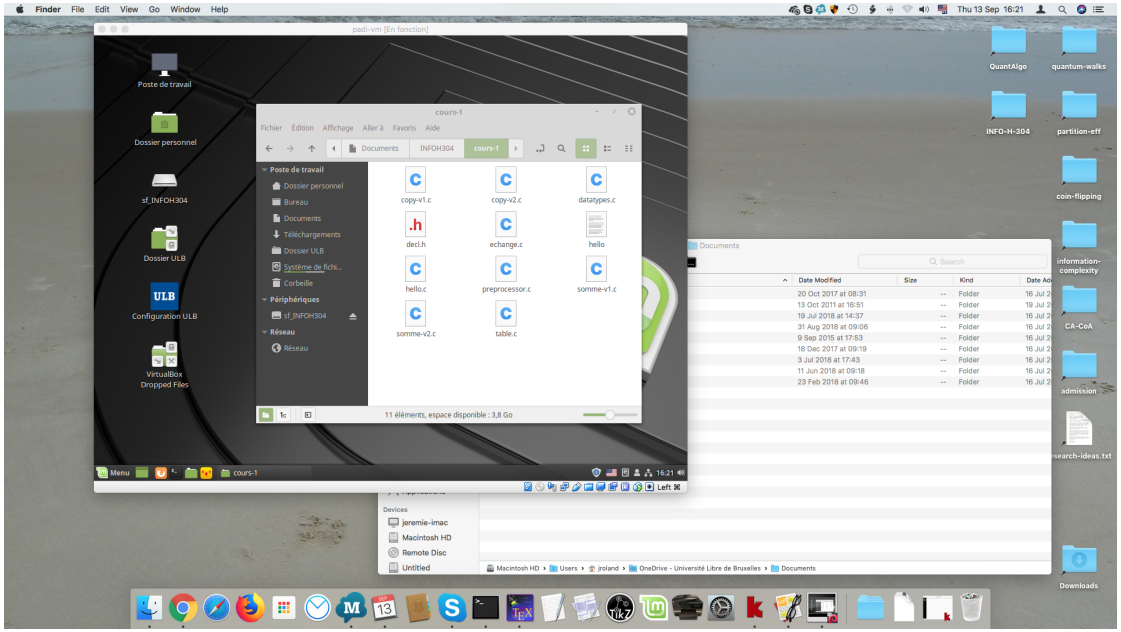
Machine virtuelle

Une machine virtuelle, c'est quoi ?

Une machine virtuelle est une simulation (ou plus exactement une émulation) d'un ordinateur (invité) par un autre ordinateur (hôte).

Exemple

- Système invité Linux sur système hôte Mac OS



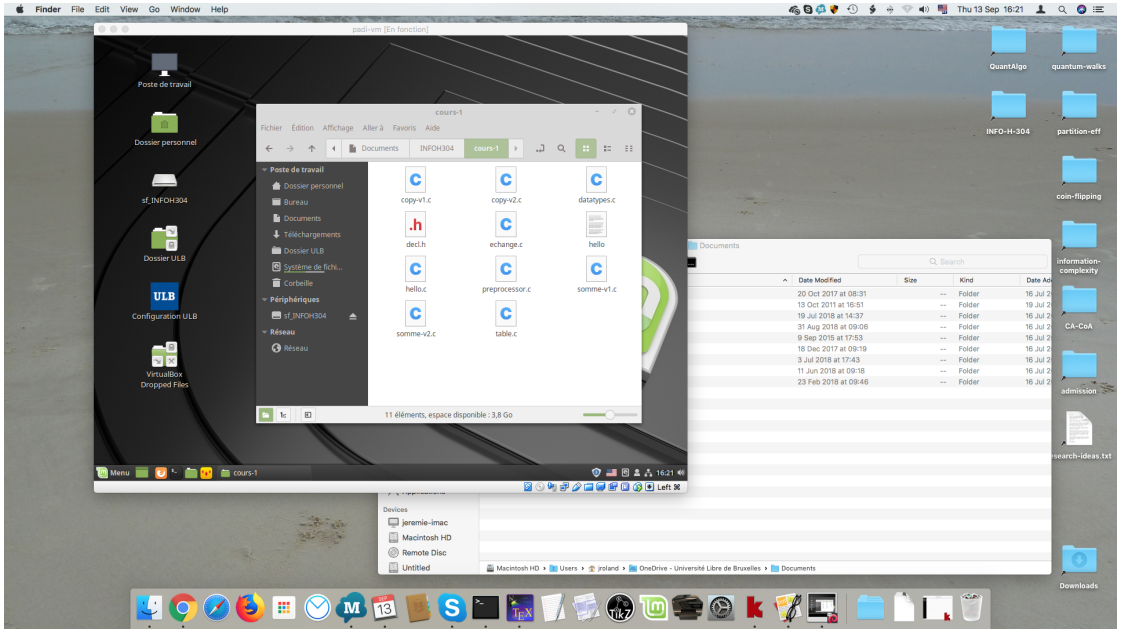
Machine virtuelle

Une machine virtuelle, pourquoi ?

- Aux cours théoriques : pour directement tester des bouts de code
- Aux TPs : pour effectuer les exercices sur votre propre ordinateur
- A la maison : pour réviser

Exemple

- Système invité Linux sur système hôte Mac OS



Installation

Démarrage rapide

1. Téléchargez et installez VirtualBox (y compris le Extension Pack)
2. Téléchargez l'image de la machine virtuelle :
   padi-vm.ova ou padi-vm.ova (lien alternatif)
3. Ouvrez ce fichier avec VirtualBox pour importer la machine virtuelle

Instructions détaillées

- Voir : Université Virtuelle
- Ou : Page web du service QuIC

Plan

Organisation du cours

Compléments de

- Programmation
  - Langage C
  - Langage C++

  } 4x2h
- Algorithmique
  - Introduction, algorithmes de tri
  - Analyse de base des algorithmes
  - Structures de données
  - Stratégies algorithmiques

  } 8x2h

- Séances d'exercices (en salle informatique) : 12x2h
- Projet : seconde moitié du quadrimestre

Charte du cours

Responsabilités partagées

- Enseignants :
  - Préparer et donner les cours de manière consciencieuse et didactique
  - Se tenir disponible pour répondre aux questions pertinentes des étudiants
  - Etre ouvert à tout commentaire ou critique constructive
- Etudiants :
  - Assister et participer activement aux cours et TPs
  - Rendre un projet respectant le cahier des charges
  - Réviser le cours et les exercices avant l'examen

Si chacun remplit sa part du contrat

- Vous aurez acquis de nouvelles compétences
- Tout en validant le cours !

Evaluation

Evaluation continue : 5 points

- Projet

Evaluation finale : 15 points

- Examen oral
  - Question de programmation (5 points)
  - Question d'algorithmique (10 points)

Attention

Pour valider le cours il faut valider l'examen oral **et** le projet

- Echec global si l'un des deux est en échec

Plan

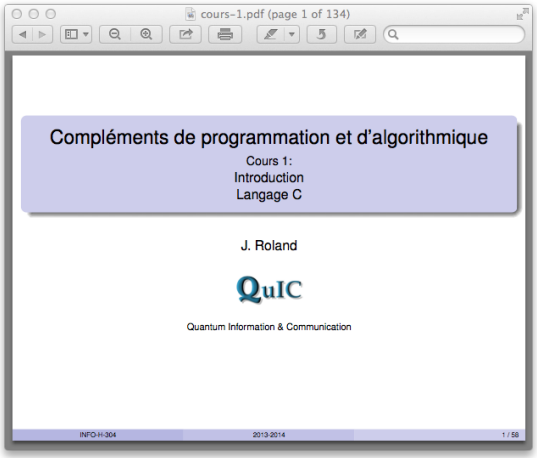
Supports de cours

Pas de syllabus mais...

De nombreux autres supports de cours

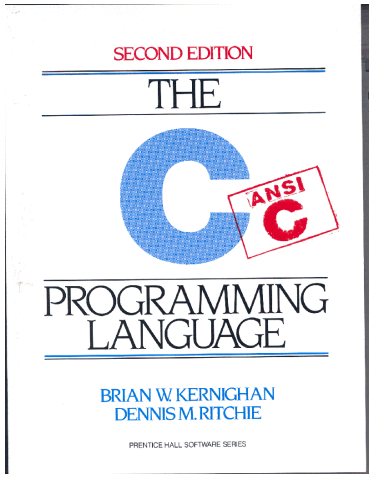
- Slides
- Livres de référence
- Ressources en ligne
- Podcast des cours et TPs
- Tutoriels vidéos
- etc.

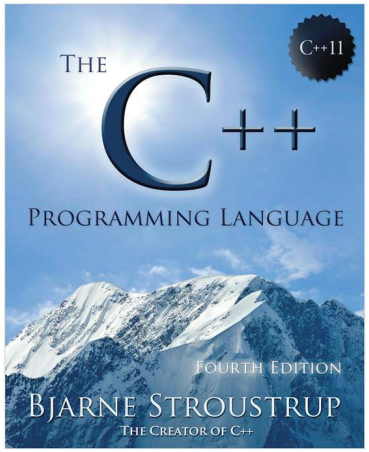
Slides

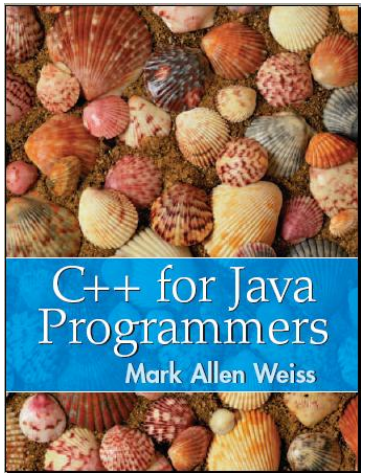


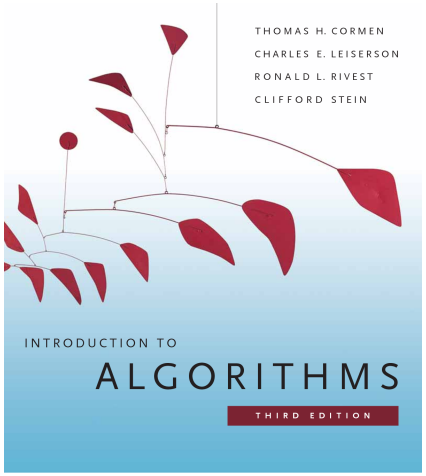
- Disponibles sur l'Université Virtuelle
- Mis en ligne sur l'UV au plus tard la veille de chaque cours
- Liste de références à la fin des slides de chaque cours :

Livres





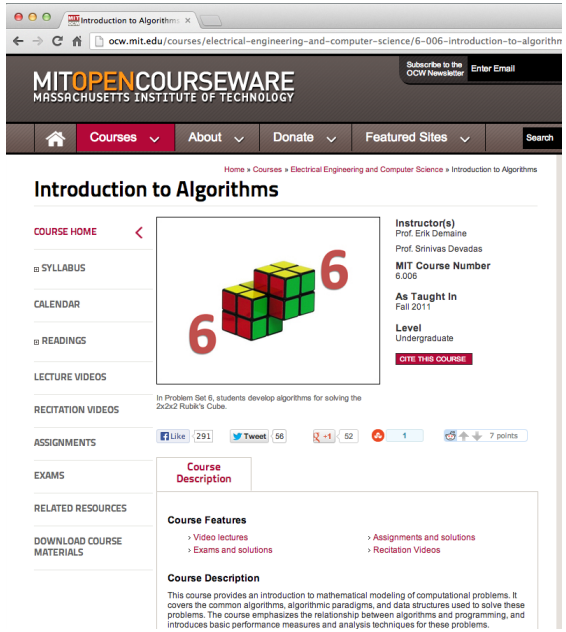




... et beaucoup d'autres

Ressources en ligne

- MIT OpenCourseWare : `http://ocw.mit.edu`



- C Programming and C++ Programming : `http://www.cprogramming.com/`
- ... et beaucoup d'autres

Plan

Bref historique

1972 Développement initial du C

- Par K. Thompson et D. Ritchie, à AT&T Bell Labs
- Intimement lié au développement de Unix

1978 “K&R C”

- 1ère édition de *The C programming language* par Kernighan et Ritchie
- Spécification informelle du langage

1989 ANSI-C ou C89

- Standardisation par la *American National Standards Institute*
- 2ème édition de *The C programming language* par Kernighan et Ritchie

1990 C90

- Certification ISO
- Equivalent à ANSI-C / C89

⋮ C99, C11, C18

Caractéristiques du C

Le C est un langage

- impératif
  - cf Python et Java
- compilé
  - cf Java
- **pas** orienté objet
  - contrairement à Python, Java,. . . et C++
- à gestion de mémoire manuelle
  - Pas de ramasse-miette (*garbage collector*) comme en Python et Java !
- simple !

Pourquoi utiliser le C ?

- Vitesse
- Mémoire
- Fonctionnalités de bas-niveau (OS, pilotes)

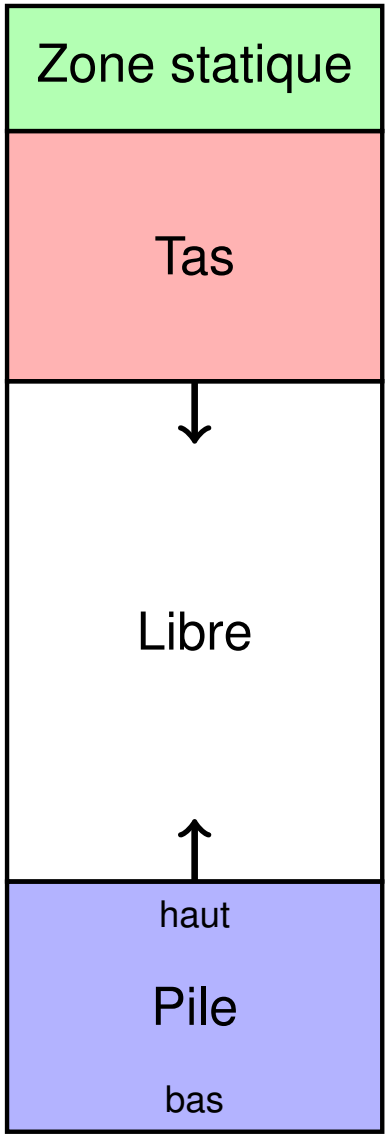
Gestion de la mémoire

Niveaux d'abstraction :

- Gestion automatique : Python, Java
- Accès à la mémoire : C, C++
- Manipulation directe de la mémoire : Assembleur

Types de mémoire :

- Zone statique :
  - Variables statiques et globales
- Mémoire pile (*stack*)
  - Variables locales
  - Remplie/vidée quand fonction lancée/terminée
- Mémoire tas (*heap*)
  - Allouée dynamiquement
  - Exemple : Objets en Java/Python



Accès à la mémoire tas

Mémoire tas :

- Peut être vue comme une grande table
- Chaque octet : adresse ↔ contenu
- Le programme entier y a accès
- Accès via “pointeurs”

Exemple : Mémoire contenant

- une chaîne de caractère “`Hello`”
- un entier `214`

Que se passe-t-il si :

- On veut allonger “`Hello`” en “`Hello, world`” ?
- On n’a plus besoin de la chaîne ?
- On lit/écrit à une adresse quelconque ?

Adresse	Contenu
⋮	⋮
0x6C0A2BF0	'H'
0x6C0A2BF1	'e'
0x6C0A2BF2	'l'
0x6C0A2BF3	'l'
0x6C0A2BF4	'o'
0x6C0A2BF5	'\0'
0x6C0A2BF6	214
0x6C0A2BF7	
⋮	⋮

Allocation dynamique de mémoire

But :

- permettre au programme
  - de désigner des bouts de mémoire comme “utilisés”
  - d’y stocker et manipuler des données
  - de libérer ces bouts de mémoire quand ils ne sont plus nécessaires

Implémentation en C

- La bibliothèque standard du C (*standard library* `stdlib.h`) fournit les fonctions
  - `malloc` : pour allouer de la mémoire
  - `free` : pour libérer cette mémoire

Plan

Le classique “Hello, world”

- Soit le fichier `hello.c`

```c
#include <stdio.h>

int main()
{
    /* Affiche le texte ``Hello, world'' */
    printf("Hello, world\n");
    return 0;
}
```

hello.c

- Compilation : `gcc -o hello hello.c`
- Execution : `./hello`

Exercice

- Compilez et exécutez ce programme
  - Quel fichier est créé par la compilation ?
  - Que se passe-t-il si on omet la ligne 1 ?

Analyse

```c
1 #include <stdio.h>
2
3 int main()
4 {
5     /* Affiche le texte ``Hello, world'' */
6     printf("Hello, world\n");
7     return 0;
8 }
```

hello.c

- Ligne 1 : Inclure le fichier `stdio.h` (bibliothèque standard)
- Ligne 3 : Fonction principale `main` à valeur de retour `int`
- Ligne 5 : Commentaire
- Ligne 6 : Affichage, requiert `stdio.h`
- Ligne 7 : Valeur de retour `0` (succès)

Types de données

Très peu de types de données de base en C

Type	Description	Mémoire [1]
`char`	caractère ASCII vu comme un entier court	1 octet
`int`	entier de taille typique pour la machine hôte	4 octets
`float`	réel à virgule flottante	4 octets
`double`	réel à virgule flottante de précision double	8 octets

```
4    char premiere = 'a';
5    int nbre_lettres = 26;
6    const float avogadro = 6.0221415e23;
7    const double pi = 3.14159265358979323846;
```

datatypes.c

Qualificateurs `short`/`long` ou `signed`/`unsigned` (voir annexe ▸) pour

- Diminuer l'utilisation de mémoire
- Augmenter ou modifier la gamme de valeurs possibles

1. Dépend de l'architecture, ici sous Mac OS 64 bits

Déclaration de variables

- Avant utilisation, une variable doit être
  - déclarée
  - éventuellement initialisée à partir d'une constante ou d'une autre variable
- Types de constantes
  - caractère : `'a'` = 97 (décimal) = `'\141'` (octal) = `'\x61'` (hexadécimal)
  - entier : 26 (décimal) = 032 (octal) = 0x1a (hexadécimal)
  - réel : 6.0221415e23 ou 3.14159265
  - expression : 24*60*60

```
4     char premiere = 'a';
5     int nbre_lettres = 26;
6     const float avogadro = 6.0221415e23;
7     const double pi = 3.14159265358979323846;
```

datatypes.c

- Mot-clé `const`
  - Empêche de modifier la valeur de la variable

Impression formatée : printf

- Exemple :

```c
#include <stdio.h>
int main()
{
    char premiere = 'a';
    int nbre_lettres = 26;
    const float avogadro = 6.0221415e23;
    const double pi = 3.14159265358979323846;
    printf("Premiere lettre de l'alphabet: %c\n", premiere);
    printf("Nombre de lettres: %10i\n", nbre_lettres);
    printf("Le nombre d'Avogadro vaut %11.3e\n", avogadro);
    printf("La constante pi vaut %10.6f\n", pi);
    return 0;
}
```

datatypes.c

- Syntaxe générale pour spécifier le format : `%[-][l][.p]s`, où
  - `[-]` permet d'aligner à gauche
  - `[l]` est la largeur minimale
  - `[.p]` est la précision
  - `s` est un symbole donnant le type (voir annexe ▸)

Fonctions

- Exemple

```
1  #include <stdio.h>
2  int somme(int a, int b);
3
4  int main()
5  {
6      int terme1 = 5, terme2 = 7;
7      int resultat = somme(terme1, terme2);
8      printf("La somme des termes vaut %i.\n", resultat);
9      return 0;
10 }
11
12 int somme(int a, int b)
13 {
14     return a+b;
15 }
```

somme-v1.c

- Doit être déclarée avant utilisation

Exercice

- Testez ce bout de code
  - Que se passe-t-il si vous omettez la déclaration en ligne 2 ? (utilisez l'option `-Wall` à la compilation)

Arguments passés par valeur

- Les arguments sont passés par valeur
- Exemple qui ne fonctionne pas :

```c
#include <stdio.h>

/* echange: tentative ratée d'échange des deux arguments */
void echange(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int premier = 1, deuxieme = 2;
    echange(premier, deuxieme);
    printf("premier vaut %i, deuxieme vaut %i.\n", premier, deuxieme);
    return 0;
}
```

echange.c

- Résultat :
  `premier vaut 1, deuxieme vaut 2.`

Variables locales

- Variables locales :
  - Arguments d'une fonction
  - Variables déclarées dans une fonction
- Deux cas possibles :
  - automatique (par défaut) : créée puis détruite à chaque exécution
  - statique (mot-clé `static`) : garde sa valeur entre les exécutions
- Portée (*scope*) : uniquement pendant l'exécution de la fonction

Variables locales

- Exemple :

```c
int somme(int a, int b)
{
    static int compteur = 0;
    int somme = a+b;
    ++compteur;
    return somme;
}
```

somme-v2.c

- La variable statique `compteur`
  - compte le nombre d'appel à `somme`
  - n'est pas accessible en dehors de `somme`

Exercice

- Reprenez le code de cette fonction et créez une fonction `main` pour la tester
  - Modifiez la fonction `somme` pour afficher la valeur de `compteur` à chaque exécution
  - Que se passe-t-il si vous essayez d'accéder à `compteur` depuis `main` ?
  - Que se passe-t-il si vous supprimez le mot-clé `static` ?

Variables globales (`extern`)

Différence entre définition et déclaration

- Déclaration : annonce les propriétés de la variable (type)
- Définition : attribue aussi un espace de stockage
  (éventuellement avec initialisation de la valeur)

- Variable globale : Variable définie en dehors de toute fonction
  - Portée : depuis sa déclaration jusqu'à la fin du fichier
  - Mot-clé `extern` : pour déclarer une variable définie dans un autre fichier
- Exemple :
  - Fichier 1 :

```
1 int variable_globale = 0;
2 int main()
3 {  ...  }
```

  - Fichier 2 :

```
1 extern int variable_globale;
2 int fonction(int a)
3 {  ...  }
```

Préprocesseur

- Les instructions commençant par # sont pour le “préprocesseur”

```c
#include <stdio.h>
#include "decl.h"
#define PI 3.14159265
#define square(x) (x)*(x)

int main()
```

preprocessor.c

- Avant compilation, le préprocesseur fait des substitutions sur le code :
  - `#include` insère les fichiers cités
    - `stdio.h` fait partie de la bibliothèque standard
    - `"decl.h"` peut contenir les déclarations de fonctions et variables globales
  - `#define` définit des macros
    - Les occurrences de PI sont remplacées par 3.14159265
    - `square(expr)` est remplacé par `(expr)*(expr)` (pour tout `expr`)

Exercice

- Testez la macro `#define square(x) (x)*(x)`
  - Pourquoi n’utilise-t-on pas la syntaxe `#define square(x) x*x` ?
  - Essayez de l’appliquer à `square(1+1)`, et interprétez le résultat

Structures de contrôle

- Comme dans la plupart des langages impératifs
  - `if-else`
  - `while`
  - `for`
  - `switch-case`
- Syntaxe : voir annexe

Opérateurs d'incrément : ++i, i++, --i, i--

- Exemple :

```c
/* table: imprime une table de multiplication */
int table(int a)
{
    int produit, facteur = 0;
    while ( facteur<10 )
    {
        produit = a*(++facteur);
        printf("%i fois %i egale %i\n", a, facteur, produit);
    }
    return 0;
}
```

table.c

- Hors d'une expression : ++i et i++ sont équivalents
  - i est incrémenté de 1, comme pour l'instruction i=i+1 ou i+=1
- Dans une expression :
  - ++i : i est incrémenté avant évaluation de l'expression
  - i++ : i est incrémenté après évaluation de l'expression
- Idem pour les opérateurs de décrément --i et i--

Opérateurs d'incrément : ++i, i++, --i, i--

- Exemple :

```
1  /* table: imprime une table de multiplication */
2  int table(int a)
3  {
4      int produit, facteur = 0;
5      while ( facteur<10 )
6      {
7          produit = a*(++facteur);
8          printf("%i fois %i egale %i\n", a, facteur, produit);
9      }
10     return 0;
11 }
12
13 int main()
14 {
15     ...
16     return 0;
17 }
```

table.c

Exercice

- Testez la fonction table
  - Que se passe-t-il si vous remplacez ++facteur par facteur++ à la ligne 7 ?

Entrée/sortie de caractères : getchar et putchar

- La commande `c=getchar();`
  - Lit un caractère sur l'entrée standard
  - Affecte ce caractère à la variable `c` de type `char`
- La commande `putchar(c);`
  - Affiche le caractère `c` sur la sortie standard
- Nécessitent `<stdio.h>`
- Exemple :

```c
#include <stdio.h>
/* Copie l'entrée sur la sortie (version 1) */
int main()
{
    char c;
    c = getchar();
    while ( c != EOF )
    {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

copy-v1.c

Exemple 1 : copier l'entrée sur la sortie

- Exemple :

```c
#include <stdio.h>
/* Copie l'entrée sur la sortie (version 1) */
int main()
{
    char c;
    c = getchar();
    while ( c != EOF )
    {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

copy-v1.c

- Analyse : EOF
  - caractère spécial signalant la fin de fichier (*End Of File*)
  - correspond généralement à l'entier -1
  - défini dans <stdio.h>
  - Peut-être entré avec Ctrl-D (Linux et Mac OS) ou Ctrl-Z (Windows)

Exemple 1 : copier l'entrée sur la sortie

- En C, toute instruction renvoie une valeur
  - L'affectation (a=b) renvoie la valeur b.
- Applications
  - Au lieu de a=0;b=0;, on peut écrire a=(b=0);, ou même a=b=0;.
  - (c=getchar()) renvoie la valeur c, et donc :

```c
#include <stdio.h>
/* Copie l'entrée sur la sortie (version 2) */
int main()
{
    char c;
    while ( (c = getchar()) != EOF )
        putchar(c);
    return 0;
}
```

copy-v2.c

Exercice

- Testez l'une ou l'autre version de ce programme
  - Observez à quel moment la copie est effectuée

Exemple 2 : compter les caractères

- Version 1, avec `while`

```c
#include <stdio.h>
/* Compte les caractères (version 1) */
int main()
{
    int nc = 0;
    while ( getchar() != EOF )
        ++nc;
    printf("%i caractères ont été entrés.\n", nc);
    return 0;
}
```

- Version 2, avec `for`

```c
#include <stdio.h>
/* Compte les caractères (version 2) */
int main()
{
    int nc;
    for ( nc=0; getchar()!=EOF ; ++nc )
        ;
    printf("%i caractères ont été entrés.\n", nc);
    return 0;
}
```

  - Le corps de la boucle est vide !

Prochain cours

Prochain cours :

Programmation en C (suite)

- Pointeurs
- Allocation manuelle de mémoire

Références

- Ces slides : disponibles sur l'Université Virtuelle
- *The C programming language*, B.M. Kernighan and D.M. Ritchie, Prentice Hall (1988)
  - Chapitres 1 à 4 et 7
- *6.088 Introduction to C Memory Management and C++ Object-Oriented Programming*, E. Kang and J. Yang, MIT OpenCourseWare (2010) : `http://ocw.mit.edu/...`
  - Lecture 1
- Wikipedia :
  - `http://en.wikipedia.org/wiki/C_(programming_language)`

Qualificateurs : `short`/`long`, `signed`/`unsigned`

Qualificateurs `short`/`long` ou `signed`/`unsigned` pour

- Diminuer l'utilisation de mémoire
- Augmenter ou modifier la gamme de valeurs possibles

Type	Gamme de valeurs[2]	Mémoire[2]
`short int` = `short`	$-2^{15} \rightarrow 2^{15}-1$	2 octets
`int`	$-2^{31} \rightarrow 2^{31}-1$	4 octets
`long int` = `long`	$-2^{63} \rightarrow 2^{63}-1$	6 octets
`long double`	$\sim 10^{-37} \rightarrow 10^{37}$	16 octets
`signed char`	$-128 \rightarrow 127$	1 octet
`unsigned char`	$0 \rightarrow 255$	1 octet
`signed int` = `int`	$-2^{31} \rightarrow 2^{31}-1$	4 octets
`unsigned int`	$0 \rightarrow 2^{32}-1$	4 octets

2. Dépend de l'architecture, ici sous Mac OS 64 bits

Impression formatée : `printf`

- Syntaxe générale pour spécifier le format : `%[-][l][.p]s`, où
  - `[-]` permet d'aligner à gauche
  - `[l]` est la largeur minimale
  - `[.p]` est la précision
  - `s` est un symbole donnant le type

Symbole	Type	Description	Exemple
`c`	`char`	caractère unique	`%c` → `"a"`
`d`, `i`	`int`	entier décimal	`%4i` → `"  26"`
`u`	`int`	entier décimal non-signé	`%4u` → `"  26"`
`o`	`int`	entier octal	`%4o` → `"  32"`
`x`, `X`	`int`	entier hexadécimal	`%4x` → `"  1a"`
`f`	`float`, `double`	réel	`%10.6f` → `"  3.141593"`
`e`, `E`	`float`, `double`	réel (notation scientifique)	`%11.3e` → `"  6.022e+23"`

Exercice

- En vous basant sur les bouts de code des derniers slides
  - Déclarez, initialisez et imprimez différents types de variables
  - Testez différents formats d'affichage pour entiers et réels
  - Que se passe-t-il si vous essayez de modifier une variable `const` ?

Structure de contrôle : `if-else`

- Syntaxe :

```
1 if (expr1)
2     instr1;
3 else
4     instr2;
```

- Pas de variables booléennes : `expr1` est un entier
  - `expr1`$=0$ $\rightarrow$`faux`
  - `expr1`$\neq 0$ $\rightarrow$`vrai`
- Exemple :

```
1 /* max: calcule le maximum des deux arguments */
2 int max(int a, int b)
3 {
4     if (a>b)
5         return a;
6     else
7         return b;
8 }
```

Autre solution : (? :)

- Syntaxe : (expr1 ? expr2 : expr3)
- Principe :
  - expr1=vrai → évalue expr2
  - expr1=faux → évalue expr3
- Exemple :

```
1 /* max: calcule le maximum des deux arguments (version 2) */
2 int max(int a, int b)
3 {
4     return (a>b ? a : b);
5 }
```

Variantes : `else-if`, `switch`

- `else if`

```
1 if (expr1)
2     instr1;
3 else if (expr2)
4     instr2;
5 else
6     instr3;
```

- `switch`

```
1 switch (expr1)
2 {
3     case const1: instr1;
4     case const2: instr2;
5     case const3: instr3;
6     default: instr4;
7 }
```

- Note :
  - Tous les `case` sont considérés, même si un `case` précédent est vrai
  - Instruction `break` pour sortir immédiatement du `switch`

Structure de contrôle : `while`

- Syntaxe :

```
1  while (expr1)
2  {
3      instructions;
4  }
```

- Exemple :

```
1  /* table: imprime une table de multiplication */
2  int table(int a)
3  {
4      int produit, facteur = 0;
5      while ( facteur<10 )
6      {
7          produit = a*(++facteur);
8          printf("%i fois %i egale %i\n", a, facteur, produit);
9      }
10     return 0;
11 }
```

Structure de contrôle : do-while

- Syntaxe :

```
1  do
2  {
3      instructions;
4  }
5  while (expr1);
```

- Exemple :

```
1  /* table: imprime une table de multiplication (version 2) */
2  int table(int a)
3  {
4      int produit, facteur = 0;
5      do
6      {
7          produit = a*(++facteur);
8          printf("%i fois %i egale %i\n", a, facteur, produit);
9      }
10     while ( facteur<10 );
11     return 0;
12 }
```

- Utilisé quand la boucle est exécutée au moins une fois

Structure de contrôle : `for`

- Syntaxe :

```
1 for (expr1; expr2; expr3)
2 {
3     instructions;
4 }
```

- Equivalent à :

```
1 expr1;
2 while (expr2)
3 {
4     instructions;
5     expr3;
6 }
```

- Utilisé en présence d'une initialisation et d'un incrément simple
- Exemple : `for (i=0;i<n;i++)`
- Différence entre version `for` et `while` : instruction `continue`
  - Un `continue` dans la boucle fait sauter à l'itération suivante
  - Pour `for`, `expr3` est tout de même exécutée

Structure de contrôle : for

- Syntaxe :

```
1  for (expr1; expr2; expr3)
2  {
3      instructions;
4  }
5  while (expr1);
```

- Exemple :

```
1  /* table: imprime une table de multiplication (version 3) */
2  int table(int a)
3  {
4      int facteur;
5      for ( facteur=1; facteur<=10; facteur++ )
6      {
7          printf("%i fois %i egale %i\n", a, facteur, a*facteur);
8      }
9      return 0;
10 }
```