Inf 431 – Cours 14

Programmation en C

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX,
01 69 33 34 67

www.enseignement.polytechnique.fr/informatique

Plan

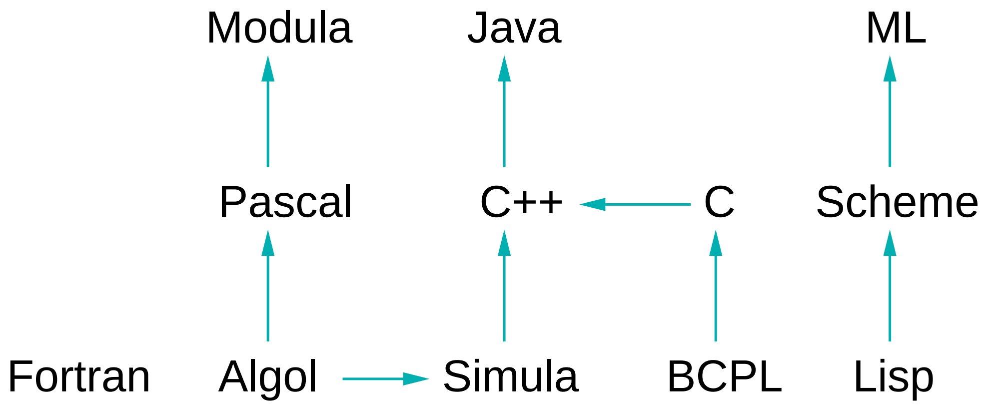
1. De Java à C
2. Caractères et entrées-sorties
3. Tableaux et pointeurs
4. Enregistrements
5. Allocation manuelle

Bibliographie

B.W. Kernighan, D.M. Rithchie, *The C Programming Language*, Prentice Hall, 1988.

B.W. Kernighan, R. Pike, *The Practice of Programming*, Addison Wesley, 1999.

Langages de programmation



- Langages typés ou non typés.
- Gestion automatique de la mémoire.
- Modularité. Programmation par objets.
- Primitives pour la concurrence.

Premier programme

```c
#include <stdio.h>

int main ()
{
  printf ("Bonjour\n");
  printf ("les élèves du cours 431\n");
}
```

A mettre dans un fichier bonjour.c, ensuite compiler
`cc bonjour.c`

et exécuter le fichier *binaire* généré
`a.out`

Par défaut, le nom du fichier *binaire* généré est a.out. On peut le choisir en faisant
`cc -o bonjour bonjour.c`

L'exécution est alors faite par
`bonjour`

De Java

```java
class Additionneur {
  public static void main (String[ ] args) {
    if (args.length != 2)
      System.out.println ("Mauvais nombre d'arguments.");
    else {
      int n1 = Integer.parseInt (args[0]);
      int n2 = Integer.parseInt (args[1]);
      System.out.println (n1 + n2);
} } }
```

à C

```c
#include <stdio.h>

int main(int argc, char *argv[ ])
{
  if (argc != 3)
    printf ("Mauvais nombre d'arguments.\n");
  else {
    int n1 = atoi(argv[1]);
    int n2 = atoi(argv[2]);
    printf ("%d\n", n1 + n2);
} }
```

De Java

```
class Syracuse {
  public static void main (String[ ] args) {
    int a = Integer.parseInt (args[0]);
    System.out.println (a);
    while (a != 1) {
      if (a % 2 == 0)
        a = a / 2;
      else
        a = 3 * a + 1;
      System.out.println (a);
} } }
```

à C

```
int main(int argc, char *argv[ ])
{
    int a = atoi (argv[1]);
    printf ("%d\n", a);
    while (a != 1) {
      if (a % 2 == 0)
        a = a / 2;
      else
        a = 3 * a + 1;
      printf ("%d\n", a);
    }
}
```

Premières remarques

- la syntaxe de Java vient de celle de C
- `argc` est le nombre de mots sur la ligne de commande
- `argv[0]` est le nom de la commande
- `argv[i]` sont les arguments de la commande pour $1 \leq i < argc$
- `atoi` (*Ascii To Integer*) convertit les chaînes de caractères en entier
- `printf` fait une impression formattée de l'argument

  `%d` en décimal

  `%f` en flottant

  `%c` en caractère

  `%s` comme une chaîne de caractères
- `#include <stdio.h>` insère au début du programme toutes les définitions nécessaires pour faire les entrées-sorties standards (*standard I/O*).
- Dans Syracuse, `a.out` (sans argument) donne comme résultat `Segmentation fault (core dumped)`. Pourquoi ?

Caractères et entrées-sorties

```c
#include <stdio.h>

int main ()
{
  int c = getchar();

  while (c != EOF) {
    putchar (c);
    c = getchar();
  }
}
```

```c
#include <stdio.h>

int main ()
{
  int c;

  while ((c = getchar()) != EOF)
    putchar (c);
}
```

Une variable de type caractère char est un entier sur 8 bits.

`getchar` retourne un entier int sur 32 bits. La valeur `EOF` est en dehors de l'intervalle $[0, 255]$ de valeurs retournées pour un caractère normal. Elle signifie qu'on vient de lire la fin de fichier.

Comme la valeur d'une affectation `x = e` est la valeur affectée à x, on a plutôt toujours tendance à écrire le programme de droite.

Caractères et entrées-sorties

```
#include <stdio.h>

int main ()
{
  int c, nLignes = 0;

  while ((c = getchar()) != EOF)
    if (c == '\n')
      ++nLignes;
  printf("%d\n", nLignes);
}
```

On compile ce programme rangé dans le fichier `lignes.c`

```
cc -o lignes lignes.c
```

On exécute le fichier *binaire* généré

```
lignes
```

On peut aussi rediriger l'entrée (grâce au shell Unix !)

```
lignes <lignes.c
```

Tableaux

En C, la taille des tableaux est fixée à la compilation. Il n'y a pas d'opérateur new pour initialiser leur valeur dynamiquement.

```
int main()
{
  int c, i;
  int nChiffre[10], nBlancs = 0, nAutres = 0;
  for (i = 0; i < 10; ++i)
    nChiffre[i] = 0;
  while ((c = getchar()) != EOF)
    if ('0' <= c && c <= '9')
      ++nChiffre[c - '0'];
    else if (c == ' ' || c == '\t' || c == '\n')
      ++nBlancs;
    else
      ++nAutres;
  printf ("chiffres =");
  for (i = 0; i < 10; ++i)
    printf (" %d", nChiffre[i]);
  printf (", blancs = %d, autres = %d\n", nBlancs, nAutres);
}
```

Pointeurs

La grande différence entre C et Java provient de l'existence de pointeurs, que l'on doit déréférencer manuellement.

Par exemple, si on considère trois variables entières

```
int x, y, z;
```

On peut considérer leurs adresses `&x`, `&y`, `&z` et les ranger dans trois pointeurs p, q, r sur des entiers

```
int *p, *q, *r;
p = &x; q = &y; r = &z;
```

On peut aussi prendre l'adresse d'un élément de tableau

```
int *p, a[10];
p = &a[3];
```

En C, on adopte la convention suivante $\boxed{a \equiv \&a[0]}$

Pour obtenir la valeur pointée par p, on écrit

```
int t;
t = *p;
```

Donc $\boxed{x \equiv *\&x}$

Pointeurs (2)

```
void echange (int *p, int *q)
{
  int x = *p;
  *p = *q;
  *q = x;
}

int main()
{
  int x = 2, y = 3;
  printf ("x = %d, y = %d\n", x, y);
  echange (&x, &y);
  printf ("x = %d, y = %d\n", x, y);
}
```

En C, les déclarations diffèrent légèrement de celles de Java.

- $t \;\; x[n]$; pour un tableau de type t
- $t \;\; x[m][n]$; pour une matrice de type t
- $t \;\; *x$; pour un pointeur sur le type t

Pointeurs et tableaux

```
void triSelection(int a[ ], int n)
{
  int i, j, imin;

  for (i = 0; i < n - 1; ++i) {
    imin = i;
    for (j = i+1; j < n; ++j)
      if (a[j] < a[imin])
        imin = j;
    echange (&a[i], &a[imin]);
  }
}

int main()
{
  int a[10] = {2, 3, 4, 1, 8, 5, 6, 9, 7, 0};
  triSelection (a, 10);
}
```

On doit passer la taille du tableau en argument, puisque la valeur de a passée en argument n'est qu'un pointeur vers son premier élément.

Arithmétique sur les pointeurs

En C, on s'amuse à faire de l'arithmétique sur les pointeurs (! ?) grâce à l'identité suivante

$$\boxed{a + i \equiv \&a[i]}$$

Donc si $p = \&a[i]$ et si on fait $++p$, alors $p = \&a[i+1]$ quel que soit le type ou la taille du tableau a. Les programmes C sont souvent truffés d'instructions `*p++` peu recommandables.

De l'identité précédente on déduit $\boxed{*(a + i) \equiv a[i]}$

Soit en ne considérant que les types dans cette nouvelle identité

$$\boxed{t \quad *a \equiv t \quad a[\,]}$$

pour tout type t.

On pouvait donc donner la signature suivante à *triSelection*

```
void triSelection(int *a, int n) { ... }
```

Chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères terminés par le caractère nul ('\0').

```c
int strlen (char s[ ])
{
  int i = 0;
  while (s[i] != '\0')
    ++i;
  return i;
}
```

```c
int strlen (char *s)
{
  int s0 = s;
  while (*s++ != '\0')
    ;
  return s - s0 - 1;
}
```

```c
int atoi (char s[ ])
{
  int r = 0; int i = 0;
  while (s[i] != '\0') {
    r = (s[i] - '0') * 10 + r;
    ++i;
  }
  return r;
}
```

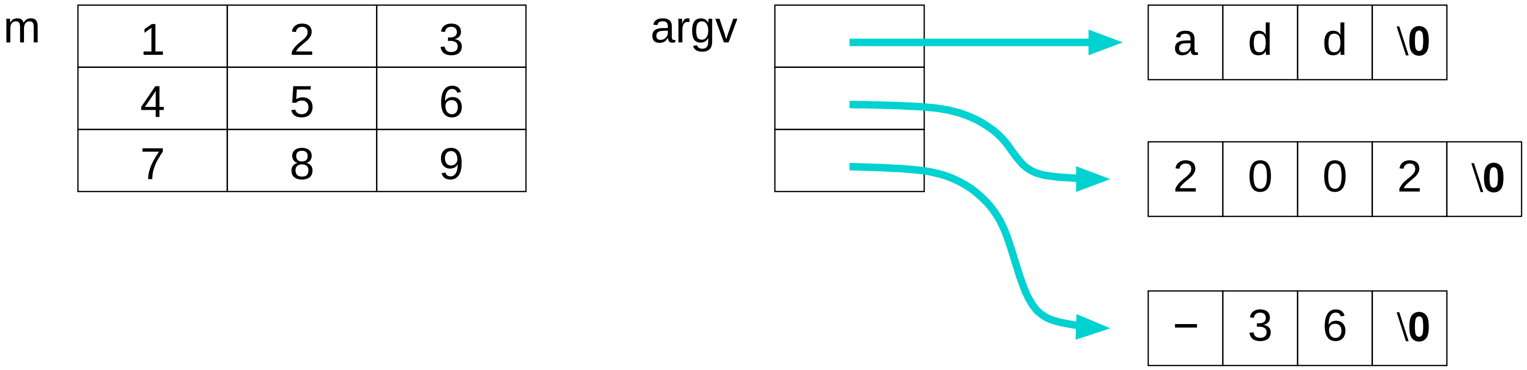
```c
int atoi (char *s)
{
  int r = 0; char c;
  while ((c = *s++) != '\0')
    r = (c - '0') * 10 + r;
  return r;
}
```

Les programmes de droite sont du C peu recommandable.

Tableaux à plusieurs dimensions

Il y a une subtilité sur les tableaux à plusieurs dimensions, où les identités précédentes ne peuvent être comprises qu'en considérant l'implantation mémoire.

```
int m[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
char *argv[3] = {"add", "2002", "-36"};
```



Enregistrements

En C, les données composites sont réalisées par la déclaration

```
struct point {
  int x;
  int y;
};

struct point p = {20, 100}, q = {40, 150};

struct point milieu (struct point p, struct point q)
{
   struct point m;
   m.x = (p.x+q.x)/2;
   m.y = (p.y+q.y)/2;
   return m;
}
```

Définition de type

Le mot-clé `typedef` permet de raccourcir l'écriture en donnant un nom aux types.

```
struct point {
  int x;
  int y;
};
typedef struct point Point;

Point p = {20, 100}, q = {40, 150};

Point milieu (Point p, Point q)
{
   Point m;
   m.x = (p.x+q.x)/2;
   m.y = (p.y+q.y)/2;
   return m;
}
```

Création de données dans le tas

Le new de Java n'existe pas en C. Mais on peut créer des données dynamiques en allouant manuellement un bout de mémoire en précisant sa taille en octets.

```
struct cellule {
  int val;
  struct cellule *suivant;
};
typedef struct cellule *Liste;

Liste nouvelleListe (int x, Liste a)
{
  Liste r = (Liste) malloc (sizeof(struct cellule));
  if (r == NULL) exit(1);
  r->val = x;
  r->suivant = a;
  return r;
}

Liste ajouter (Liste a, Liste b) {
  if (a == NULL) return b;
  else return nouvelleListe(a->val, ajouter(a->suivant, b));
```

Quelques remarques

- Contrairement à Java, les données composites (tableaux, enregistrements) peuvent être rangées dans la zone des données automatiques (*la pile*).
- On peut prendre l'adresse de toute donnée et la ranger dans un pointeur.
- p **->** x est un raccourci pour $(*p).x$
- l'obsession de C est de ne perdre aucun octet.
- sizeof donne la taille en octets de toute donnée.
- $\mathit{malloc}(n)$ (*memory allocate*) alloue un bloc contigu de n octets dans la zone des données dynamiques (*le tas*). La valeur retournée est un pointeur de type void* qu'on doit convertir en pointeur du bon type.
- $\mathit{free}(p)$ libère le bloc pointé par p précédemment alloué par malloc.

Pile et tas

Il y a 2 types de données en C (comme en Java) :

- les variables globales ou les données allouées dans la pile,

```
Point p = {20, 100}, q = {40, 150};

Point milieu (Point p, Point q)
{
   Point m;
   m.x = (p.x+q.x)/2;
   m.y = (p.y+q.y)/2;
   return m;
}
```

- les données dynamiques allouées dans le tas (*heap*).

```
Liste nouvelleListe (int x, Liste a)
{
  Liste r = (Liste) malloc (sizeof(struct cellule));
  r->val = x;
  r->suivant = a;
  return r;
}
```

Les données dans la pile vivent le temps des appels des fonctions dont elles sont des variables locales, Les données dynamiques restent jusqu'à une libération manuelle par *free*.

Glaneurs de cellules – GC

En Java, tous les pointeurs pointent sur le tas. Les variables sont typées. On distingue clairement une variable contenant une adresse par son type (objet, tableau).

En C, la situation est moins claire, puisqu'on peut prendre l'adresse de toute variable et ranger sa valeur dans une autre. Pire, on peut faire des conversions entre types sans aucun contrôle.

En Java, on peut retrouver facilement tous les pointeurs à partir des variables allouées dans la pile, parcourir le graphe des objets accessibles depuis la pile, et libérer les objets inaccessibles pour pouvoir les allouer par la suite. C'est ce que fait le glaneur de cellules (GC) (*garbage collector*).

De nombreux algorithmes existent pour le GC (compteurs de références, par traçage, par recopie, incrémental, conservatif, etc)

L'implémentation des langages de programmation est étudiée dans le cours Langages de Programmation, majeure 1, et Compilation, majeure 2.

Préprocesseur

Avant toute compilation, le préprocesseur change le source du programme. Pour les curieux `cc -E programme.c` donne le résultat.

- `#include <fichier.h>` insère un fichier de définitions de signatures de fonctions. Ces fichiers figurent souvent dans le répertoire `/usr/include`.
- On peut remplacer toutes les occurences de `VRAI` par 1 ou de `FAUX` par 0. On peut aussi donner des paramètres à ces définitions de macros.

```
#define VRAI 1
#define FAUX 0
#define Max(x,y) ((x) > (y) ? (x) : (y))
```

Il y a plein d'autres directives, toutes commençant par `#`, notamment pour tenir compte du type du processeur ou de la version du système d'exploitation.

Exercices

Exercice 1 En C, il n'y a pas de type booléen (grave erreur !). La constante `FAUX` est représentée par l'entier 0 ; et `VRAI` est représentée par tout entier non nul. Quel est le sens de if `(x = 0)` S else S' ? Comparer au sens de if `(x == 0)` S else S'.

Exercice 2 Quel est le sens de `*p-- = *++p` ?

Exercice 3 Comparer les valeurs de `strlen("Bonjour")` et de `sizeof("Bonjour")` ?

Exercice 4 On suppose que `rang(m)` calcule le rang de la matrice m. Expliquer qu'on ne peut écrire la déclaration sous la forme int `rang(`int `m[ ][ ])`, mais int `rang(`int `m[`n`][ ])` où n est la première dimension.

Exercice 5 Dans les GCs par compteurs de références, on adjoint à chaque cellule allouée dans le tas un compteur du nombre d'objets ou autres données qui pointent sur cette cellule. Comment gérer cette information ? Essayer de deviner le fonctionnement d'un tel GC.