UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Programmation avancée et Complexité

Chapitre 2 – De Python au langage C

Frédéric Flouvat

Basé sur le cours "C for Python Programmers" de C. Burch (Hendrix College) et E. Patitsas (University of Toronto)

Université de la Nouvelle-Calédonie

frederic.flouvat@univ-nc.nc

Plan

- De Python au langage C
  - Les bases
  - Les principales opérations et instructions
  - Les librairies
  - Les pointeurs

Les bases

Un petit peu d'histoire sur le C

- Créé dans les années 70 par Ken Thompson (Bell Lab.) pour aider au développement du système d'exploitation UNIX.
  - Conçu pour pouvoir accéder à toutes les fonctionnalités des systèmes UNIX
- Le premier langage de développement dans les années 80.
- A influencé de nombreux langages de programmation tels que C++, C#, Java, JavaScript, etc.

Et maintenant ?

- Toujours un langage très utilisé
  - Au coeur des principaux systèmes d'exploitation (Windows, Linux, Mac, Android et iOS);
  - Au coeur des principaux Systèmes de Gestion de Bases de Données (p.ex. Oracle, SQL Server et PostgreSQL);
  - Au coeur des systèmes embarqués et objets connectés.

- Pourquoi cette popularité ?
  - Possibilité de manipuler très finement la mémoire (lectures et écritures);
  - Une utilisation contrôlée des ressources (p.ex. libération de la mémoire);
  - La taille du code généré;

Compilateur versus Interpréteur

- Le compilateur en C
  - Traduit le code (fichier texte) en langage machine (fichier exécutable)
  - Les étapes à suivre:
    - écrire un fichier texte (extensions .c ou .h) contenant le programme en langage C
    - compiler le "code source" en "exécutable"
    - lancer le fichier exécutable

- L'interpréteur en Python
  - Lit le code source et l'exécute directement
  - Les étapes à suivre:
    - écrire un fichier texte (extension .py) contenant le programme en langage C
    - passer ce code source en paramètre de l'interpréteur

➢ Un fonctionnement très différent
  - Une étape en moins pour le Python
  - Un exécutable en général beaucoup plus performant en C

Exemple : mon premier programme en C

Toute instruction en C doit être définie à l'intérieur d'une **fonction/procédure**

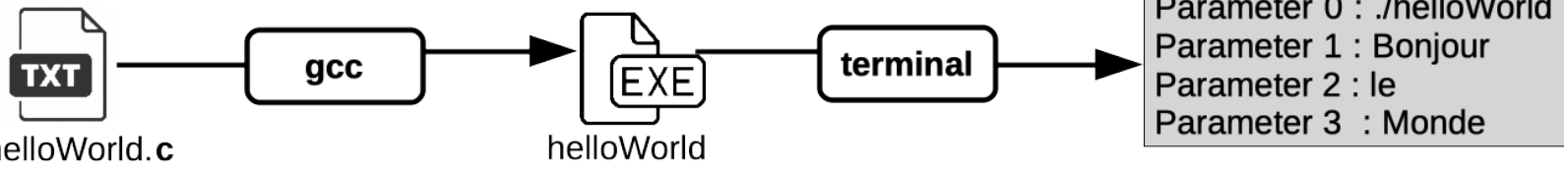
helloWorld.c

```
#include <stdio.h>

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

      int i = 0 ;

      while ( i < argc ) {
            printf("Parameter %d : %s \n", i, argv[i] );
            i++;
      }
      return 0 ;
}
```

Le point d'entrée à l'exécution est la fonction **main**( )

Compilation :
gcc -o helloWorld helloWorld.c

Exécution :
./helloWorld Bonjour le Monde



helloWorld.**c** helloWorld

La déclaration des variables

- Obligation d'indiquer au compilateur les variables avant qu'elles soient utilisées

- En C, déclaration d'une variable = définition du type de la variable
  - Syntaxe : *typeName variableName* ;
  - p.ex.

```
int i ;
double j = 3.2 ;
```

- Impossible de changer le type de la variable après la déclaration

```
int i ;
char i ;
```

```
double j = 3 ;
```

➢ `j` à pour valeur le double 3 .0

- Variable non déclarée ➔ Erreur "symbol undeclared" à la compilation

➢ Avantage : erreur d'utilisation des variables toutes connues à la compilation
(et non au fur et à mesure de l'exécution comme en Python)

Les espaces, tabulations et retours à la ligne

- En **Python**, importants car
  - Retour à la ligne = une nouvelle instruction
  - Espace et tabulation = un bloc (i.e. regroupement) d'instructions

- En **C**, utilisés pour faire de l'indentation (mise en page du code)
  - Fortement conseillé mais non obligatoire
  - Une nouvelle instruction = point virgule **;**
  - Un bloc d'instructions = accolades ouvrante et fermante **{ … }**

Attention : code illisible

C
```
disc=b*b-4*a*c;if(disc<0){
num_sol=0;}else{t0=-b/a;if(
disc==0){num_sol=1;sol0=t0/2
;}else{num_sol=2;t1=sqrt(disc/a;
sol0=(t0+t1)/2;sol1=(t0-t1)/2;}}
```

Python
```
disc = b * b - 4 * a * c
if disc < 0:
    num_sol = 0
else:
    t0 = -b / a
    if disc == 0:
        num_sol = 1
        sol0 = t0 / 2
    else:
        num_sol = 2
        t1 = disc ** 0.5 / a
        sol0 = (t0 + t1) / 2
        sol1 = (t0 - t1) / 2
```

C
```
disc = b * b - 4 * a * c;
if (disc < 0)
{
    num_sol = 0;
}
else
{
    t0 = -b / a;
    if (disc == 0)
    {
        num_sol = 1;
        sol0 = t0 / 2;
    }
    else
    {
        num_sol = 2;
        t1 = sqrt(disc) / a;
        sol0 = (t0 + t1) / 2;
        sol1 = (t0 - t1) / 2;
    }
}
```

La fonction `printf()`

- Affichage de texte à l'écran (la sortie standard)
  - Important notamment pour débugger
  - Un fonctionnement qui peut sembler complexe

- Appartient à la librairie `stdio.h`

- Signature: `int printf ( const char * format, ... );`
  - *`format`* : chaîne de caractères indiquant le format de ce qui va être affiché
    - %d : un entier; %c : un caractère; %s : une chaîne de caractère;
  - *autres paramètres* : les valeurs à afficher
  - cf. http://www.cplusplus.com/reference/cstdio/printf/ pour plus d'informations

```
printf("Parameter");
printf("Parameter %d", 10 );
printf("Parameter %d", i );
printf("Parameter %d \n", i );
printf("Parameter %d : %s \n", i, argv[i] );
```

Les fonctions en C

- Tout code doit être dans une fonction
- Les fonctions ne peuvent être imbriquées
  - Une liste de fonctions pouvant s'invoquer entre elles

C
```c
double expon(double b, int e)
{
    if (e == 0)
    {
        return 1.0;
    }
    else
    {
        return b * expon(b, e - 1);
    }
}
```

- Syntaxe: *typeReturn functionName(type1 paremeter1, type2 parameter2, ⋯){ ⋯ }*
  - Si la fonction ne retourne rien , *typeReturn* = *void*
- Fonction utilisée si invoquée (directement ou indirectement) dans la fonction "main" (le programme principal)

Python
```python
def gcd(a, b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)

print("GCD: " + str(gcd(24, 40)))
```

équivalents ⟺

C
```c
int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

int main()
{
  printf("GCD: %d\n",
    gcd(24, 40));
  return 0;
}
```

Les principales opérations et instructions

Les opérateurs

- Un "mot-clé" du langage utilisé pour faire une opération arithmétique
  - Les principaux opérateurs :

C

```
++ -- (postfixe/préfixe)
+ - ! (unaire)
* / %
+ - (binaire)
< > <= >=
== !=
&&
||
= += -= *= /= %=
```

Python

```
**
+ - (unaire)
* / % //
+ - (binaire)
< > <= >= == !=
not
and
or
```

  - Différences :
    - pas d'opérateur puissance en C (fonction `pow()`)
    - syntaxe différente des opérateurs logiques
    - affectation : opérateur en C vs instruction en Python
      - possibilité de faire une affectation dans une instruction

```
while( (a = getchar()) != EOF )…
```

    - opérateurs ++, --, +=, -=, *=, …
    - division en C : division entière si les deux valeurs en entrée sont entières
      - 13/5 = 2

Les principaux types (1/2)

Les types entiers

Type	Taille	Valeurs
`char`	1 octet	-128 à 127 ou 0 à 255
`unsigned char`	1 octet	0 à 255
`signed char`	1 octet	-128 à 127
`int`	2 ou **4 octets***	-32 768 à 32 767 ou **-2 147 483 648 à 2 147 483 647**
`unsigned int`	2 ou **4 octets***	0 à 65 535 ou **0 à 4 294 967 295**
`short`	2 octets	-32 768 à 32 767
`unsigned short`	2 octets	0 à 65 535
`long`	8 octets	-9223372036854775808 à 9223372036854775807
`unsigned long`	8 octets	0 à 18446744073709551615

*dépend du compilateur (fonction sizeof(type) pour connaître la taille)

Les types réels

Type	Taille	Valeurs	Précision
`float`	4 octets	$3.4*10^{-38}$ à $3.4*10^{+38}$	simple précision (7 chiffres significatifs)
`double`	8 octets	$1.7*10^{-308}$ à $1.7*10^{+308}$	double précision (15 chiffres significatifs)
`long double`	10 octets	$3.4*10^{-4932}$ à $3.4*10^{+4932}$	double précision au moins

Les principaux types (2/2)

- Pas de type booléen en C
  - Utilisation d'un entier
    - 0 = *false* et 1 = *true*

```
int main()
{
    int i = 5;
    if (i)
    {
        printf("in if\n");
    }
    else
    {
        printf("in else\n");
    }
    return 0;
}
```

valide mais code difficile à lire

```
int main()
{
    int i = 5;
    if (i != 0)
    {
        printf("in if\n");
    }
    else
    {
        printf("in else\n");
    }
    return 0;
}
```

- Traités comme des entiers par les opérateurs
  - p.ex. nombre de variables positives

```
pos = (a > 0) + (b > 0) + (c > 0);
```

Remarque sur les accolades { et }

- Indique un bloque (i.e. groupe) d'instructions ➔ pas obligatoire si une seule instruction

```
if (first > second)
    max = first;
else
    max = second;
```

```
if (first > second)
{
    max = first;
}
else
{
    max = second;
}
```

```
if (disc < 0)
{
    num_sol = 0;
}
else
{
    if (disc == 0)
    {
        num_sol = 1;
    }
    else
    {
        num_sol = 2;
    }
}
```

```
if (disc < 0)
{
    num_sol = 0;
}
else
    if (disc == 0)
    {
        num_sol = 1;
    }
    else
    {
        num_sol = 2;
    }
```

```
if (disc < 0)
{
    num_sol = 0;
}
else if (disc == 0)
{
    num_sol = 1;
}
else
{
    num_sol = 2;
}
```

```
if (disc < 0)
    num_sol = 0;
else if (disc == 0)
    num_sol = 1;
else
    num_sol = 2;
```

- Conseil: laisser tout de même les accolades
  - Facilite l'ajout d'instructions
  - Facilite la lecture du code

Synthèse des principales instructions (1/2)

- Les déclarations de variable `int x;`
- Les expressions `x = y + z ;` `printf("%d", x) ;`
- Les conditions `if ( x < 0 ) { printf(" negative"); }`
- Le `return` `return 0 ;`
  - `void` si pas de valeur retournée
- Les boucles `while(`*`condition`*`) { … }`
- Les boucles `for(`*`init; test; update`*`)`

```
for (p = 1; p <= 512; p *= 2) {
    printf("%d\n", p);
}
```

```
while (i >= 0)
{
    printf("%d\n", i);
    i--;
}
```

```
for (i = 0; i < n; i++)
{
    …
}
```

Synthèse des principales instructions (2/2)

- Autre formes de conditions : `switch`
  - Remplace `if … else if … else if … else`
  - `case` : un caractère ou un entier
  - Attention : ne pas oublier les `break`
    - sinon exécution de tous les blocs

```
switch (letter_grade) {
case 'A':
    gpa += 4;
    credits += 1;
    break;
case 'B':
    gpa += 3;
    credits += 1;
    break;
case 'C':
    gpa += 2;
    credits += 1;
    break;
case 'D':
    gpa += 1;
    credits += 1;
    break;
case 'W':
    break;
default:
    credits += 1;
}
```

Les structures de données (1/2)

- Peu de structure de données directement disponibles en C (contrairement à Python)
- Les structures composées
  - Regroupement de variables

```
struct Coordonnees
{
    int x; // Abscisses
    int y; // Ordonnées
};

int main( int argc, char * argv[] )
{
    struct Coordonnees point ;
    point.x = 0 ;
    point.y = 1 ;
};
```

  - Possibilité de définir un type en combinant avec `typedef`

```
struct Coord
{
    int x; // Abscisses
    int y; // Ordonnées
};

typedef struct Coord Coordonnees ;
```

OU

```
typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;
```

```
int main( int argc, char * argv[] )
{
    Coordonnees point ;
    point.x = 0 ;
    point.y = 1 ;
};
```

Les structures de données (2/2)

- Les tableaux
  - Similaires aux listes Python mais avec une taille fixe défini à la création de la variable

```c
char vowels[6] = {'a','e','i','o','u','y'};
```

```c
double pops[50];
pops[0] = 897934;
pops[1] = pops[0] + 11804445;
```

  - Pas d'accès direct à la taille du tableau après sa création
    - pas de `len(vowels)`
    - `size_t n = sizeof(vowels) / sizeof(char);`

- Attention si utilisation d'un indice hors des limites du tableau
  - Python : arrête le programme "proprement" en indiquant l'erreur
  - C : accède à la case mémoire en question
    - pas d'erreur : le programme fonctionne avec des mauvaises valeurs
    - erreur avec arrêt du programme (*segmentation fault* ou *bus error*) mais pas forcément à l'endroit du code où l'accès a été effectué
    - très difficile à débugger

Les librairies

Les librairies et prototypes de fonction

- Organiser son code en le répartissant dans plusieurs fichiers
  - Améliore la lisibilité, la maintenance, l'évolutivité du code

- Problème : fonctions déclarées obligatoirement avant leur utilisation en C
- Comment faire quand les fonctions sont déclarées dans des fichiers différents ?
  - Un vrai problème pour le compilateur

- Les prototypes de fonction
  - Décrire uniquement l'entête de la fonction (sa signature), sans son corps

main.c

```
int gcd(int a, int b);

int main()
{
    printf("GCD: %d\n", gcd(24, 40));
    return 0;
}
```

indique au compilateur que la fonction n'a pas encore été définie

```
int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}
```

math.c

Les fichiers entêtes ou *header*

- Un fichier avec l'extension *.h* regroupant des prototypes de fonction (et la définition des structures de données)
  - Importé par les fichiers sources utilisant la fonction (`#include "…"`)
  - ➢ Evite de répéter les prototypes de fonctions dans chaque fichier source l'utilisant

main.c

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

int main()
{
    printf("GCD: %d\n", gcd(24, 40));
    return 0;
}
```

mathfun.h

```c
int gcd(int a, int b);

double expon( double  b, int e );

…
```

mathfun.c

```c
#include "mathfun.h"

int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

double expon(double b, int e)
{
    if (e == 0)
        return 1.0;
    else
        return b * expon(b, e - 1);
}

…
```

Le préprocesseur

- Programme prétraitant le code source avant de l'envoyer au compilateur
  - Langage basé sur des commandes (*directives*)

➢ Inclusion de fichiers : `#include "filename"` ou `#include <filename>`
- remplace le "*include*" par le contenu du fichier associé (i.e. les prototypes de fonctions)

➢ Définition de macros : `#define VARNAME value`
- des constantes (texte remplacé avant la compilation)

```
#define PI 3.14159
```

```
printf("area: %f\n", PI * r * r);
```

➢ Compilation conditionnelle : `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` et `#endif`
- p.ex. inclure des librairies différentes en fonction de l'OS ciblé (paramètre du compilateur)

```
#ifdef __unix__
#include <unistd.h>
#elif defined _WIN32
#include <windows.h>
#endif
```

Compiler un code réparti entre plusieurs fichiers

1. Insérer des directives de compilation conditionnelle pour éviter au compilateur d'essayer de compiler plusieurs fois le même code (*error: redefinition of function …*)

mathfun.h

```
#ifndef MATHFUN_H
#define MATHFUN_H

int gcd(int a, int b);

double expon( double  b, int e );

…
#endif
```

mathfun.c

```
#include "mathfun.h"

int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

…
```

2. Compiler chaque fichier source pour créer un fichier objet
   - gcc –c –Wall main.c
   - gcc –c –Wall mathfun.c

3. Faire l'édition des liens et construire l'exécutable final (lier les fichiers compiler pour créer l'exécutable)
   - gcc –o myprog main.o mathfun.o

Compiler un code réparti entre plusieurs fichiers

- Possibilité d'utiliser un fichier **Makefile** pour éviter de tout devoir retaper à chaque fois

makefile

```
myprog: main.o structure.o operation.o
    gcc -o prog main.o structure.o operation.o

main.o : main.c
    gcc -c -Wall main.c

structure.o : structure.c
    gcc -c -Wall structure.c

operation.o : operation.c
    gcc -c -Wall operation.c
```

- Ecrire simplement *make* dans le terminal pour tout recompiler

➢ Cf. http://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/Makefile.html pour plus de détails

Les pointeurs

Les bases

- Un pointeur = une variable représentant l'adresse mémoire vers une donnée
  - Un concept caché dans beaucoup d'autres langages mais sous jacent
- Déclaration : *typeName * variableName ;*
- Accès à l'adresse mémoire d'une variable : operateur &
- Accès à l'emplacement mémoire référencé par un pointeur : opérateur *
  - appelé aussi "déréférencement"
- Pointeur null : NULL

```
int i;
int *p;

i = 4;
p = &i;
*p = 5;
printf("%d\n", i);
```

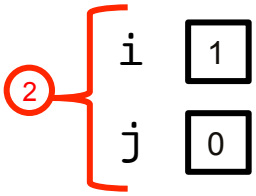
Passage par valeur, par référence et pointeurs (1/3)

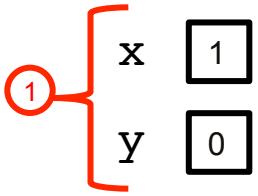
- Passage par valeur d'un paramètre d'une fonction
  - La valeur est recopiée de manière temporaire dans la fonction

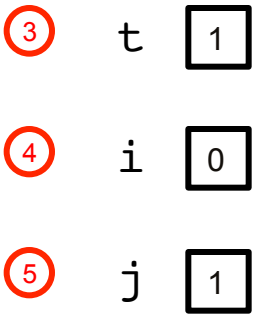
```
void swap(int i, int j) {
    int t;

3   t = i;
4   i = j;
5   j = t;
}
```

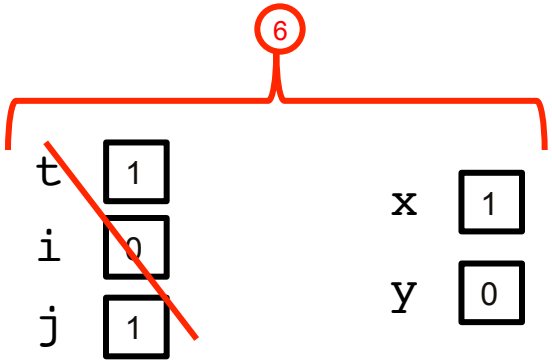
```
1   int x = 1 ;
    int y = 0;

2   swap(x, y);   6

    printf("x=%d, y=%d", x, y);
```

compile mais ne fait rien !









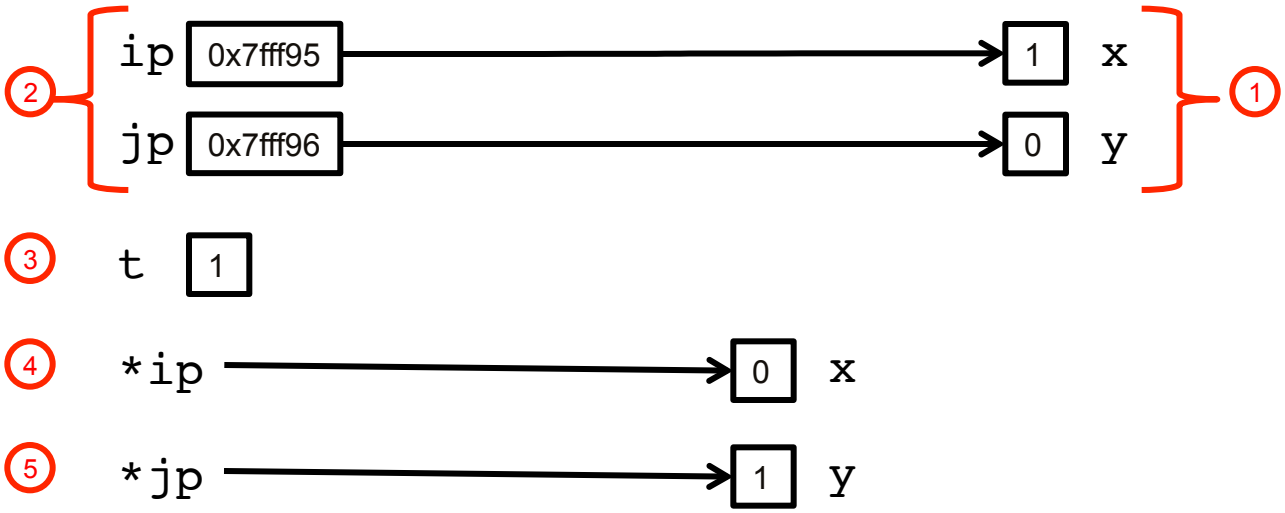
Passage par valeur, par référence et pointeurs (2/3)

- Passage par référence (pointeur) d'un paramètre d'une fonction
  - L'adresse mémoire est recopiée de manière temporaire, mais il est possible d'y accéder est de modifier durablement son contenu

```
void swap(int * ip, int * jp) {
    int t;

3   t = *ip;
4   *ip = *jp;
5   *jp = t;
}
```

```
1   int x = 1 ;
    int y = 0;

2   swap( &x, &y);

    printf("x=%d, y=%d", x, y);
```

échange les valeurs de x et y



Passage par valeur, par référence et pointeurs (3/3)

- **Autre exemple** : la fonction `scanf`
  - Fonction permettant de lire les informations saisies au clavier
  - Prend en deuxième paramètre l'adresse mémoire d'une variable pour pouvoir enregistrer la valeur lue à l'intérieur

```
printf("Type a number. ");
scanf("%d", &i);
printf("The value is %d", i );
```

- **Attention :**

```
void swap(int * ip, int * jp) {
    int * tp;

    tp = ip;
    ip = jp;
    jp = tp;
}
```

```
int x = 1 ;
int y = 0;

swap(&x, &y);

int * xp ;
int * yp ;

xp = &x ;
yp = & y ,

swap(xp, yp);

printf("x=%d, y=%d", x, y);
```

compile mais ne fait rien !

Retour sur les tableaux

- Variable de type tableau en C = un pointeur vers la première case du tableau en mémoire
  - Les autres cases sont à la suite

```
char vowels[6] = {'a','e','i','o','u','y'};
printf("%c", vowels[2] );
```

⟺

```
char vowels[6] = {'a','e','i','o','u','y'};
printf("%c", *(vowels+2) );
```

- Passer en paramètre un tableau à une fonction = passer en paramètre un pointeur

```
void setToZero(int *arr, int n) {
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = 0;
    }
}

int main() {
    int grades[50];

    setToZero(grades, 50);
    return 0;
}
```

Les chaînes de caractères (*string*)

- Pas de type prédéfini ➔un simple tableau de caractères (i.e. un pointeur)
  - Fin de la chaîne de caractères : caractère `\0`

```
char sentence[20] = "the dog is agog.";
```

t	h	e		d	o	g		i	s		a	g	o	g	.	\0			

- Copier une chaîne de caractères = copier un tableau

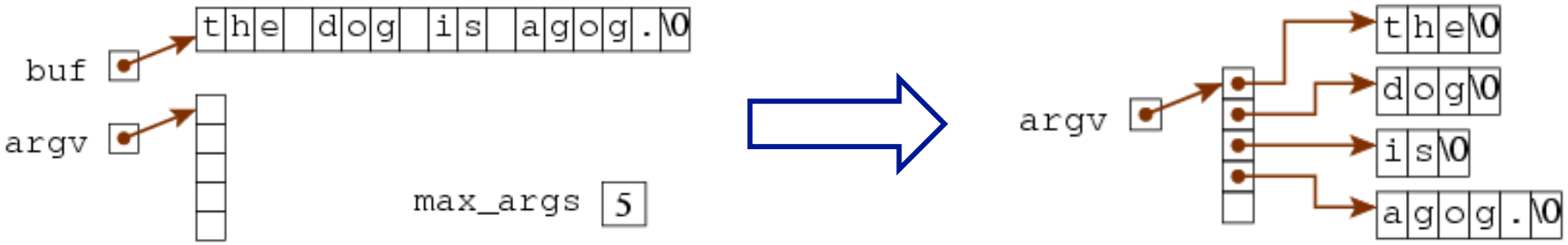
```
for (i = 0; src[i] != '\0'; i++) {
    dst[i] = src[i];
}
dst[i] = '\0';
```

- Beaucoup de fonctions déjà implémentées dans `string.h`
  - **void** strcpy(**char** *dst, **char** *src)
    - copie deux chaînes (dont le \0)
  - **int** strlen(**char** *src)
    - retourne la taille de la chaîne (sans le \0)
  - **int** strcmp(**char** *a, **char** *b)
    - retourne 0 si les deux chaînes sont identiques, <0 si a est plus petit que b et >0 si a est plus grand que b (ordre lexicographique)

Application à l'extraction des mots dans une phrase (*tokenization*)

stringFunc.h

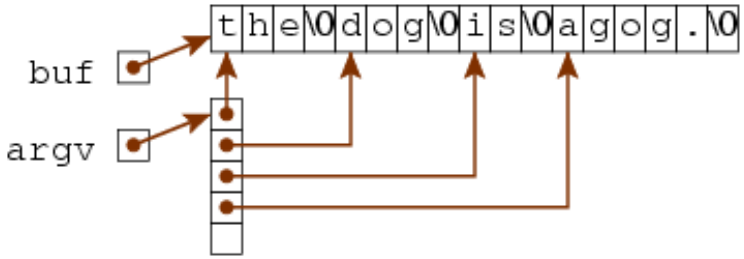
```
#indef STRINGFUNC
#define STRINGFUNC

#include <ctype.h>

/* splitLine
 *  Breaks a string into a sequence of words. The pointer to each successive word
 *  is placed into an array. The function returns the number of words found.
 *
 * Parameters:
 *  buf - string to be broken up
 *  argv - array where pointers to the separate words should go.
 *  max_args - maximum number of pointers the array can hold.
 *
 * Returns:
 *  number of words found in string.
 */
int splitLine(char *buf, char **argv, int max_args) ;

#endif
```



Application à l'extraction des mots dans une phrase (*tokenization*)

stringFunc.c

```
#include "stringFunc.h"

/* splitLine
 *  Breaks a string into a sequence of words. The pointer to each successive word
 *  is placed into an array. The function returns the number of words found.
 *
 * Parameters:
 *  buf - string to be broken up
 *  argv - array where pointers to the separate words should go.
 *  max_args - maximum number of pointers the array can hold.
 *
 * Returns:
 *  number of words found in string.
 */
int splitLine(char *buf, char **argv, int max_args) {
    int arg;

    /* skip over initial spaces */
    while (isspace(*buf)) buf++;

    for (arg = 0; arg < max_args && *buf != '\0'; arg++) {
      argv[arg] = buf;

      /* skip past letters in word */
      while (*buf != '\0' && !isspace(*buf)) {
        buf++;
      }

      /* if not at line's end, mark word's end and continue */
      if (*buf != '\0') {
        *buf = '\0';
        buf++;
      }

      /* skip over extra spaces */
      while (isspace(*buf)) buf++;
    }
    return arg;
}
```



Structures de données et pointeurs

- Souvent combinés
  - Pointeur sur une structure de données
  - Tableau de structure de données

```
typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;
```

```
int main() {
    Coordonnees c;

    c.x = 50;
    c.y = 100;

    double dist = distToOrigin( &c );

    return 0;
}
```

➢ Peut éviter de recopier beaucoup de valeurs

```
#include <math.h>

double distToOrigin(struct Coordonnees *p) {
    return sqrt( (*p).x * (*p).x + (*p).y * (*p).y) );
}
```

- `(*p).x` peut aussi être écrit `p->x`

```
#include <math.h>

double distToOrigin(struct Coordonnees *p) {
    return sqrt( p->x * p->x + p->y * p->y );
}
```

Allocation dynamique de la mémoire (1/2)

- Permet d'allouer/libérer la mémoire associée à une variable au moment de l'exécution en fonction des données ou des actions des utilisateurs

- Allocation : fonction `void* malloc (size_t size)`
  - `size` : entier représentant la taille en octet de la mémoire à allouer
    - utiliser la fonction `size_t` **`sizeof`**`( typeName )`

- Libération : fonction `void free(void *ptr)`
  - `ptr` : adresse mémoire de la variable à libérer

```c
#include "struct.h"

int main() {
    int n;
    Coordonnees *arr;

    printf("How many points ? ");
    scanf("%d", &n);
    arr = (Coordonnees *) malloc( n * sizeof(Coordonnees) );

    …
    free( arr ) ;
    return 0;
}
```

struct.h

```c
#ifndef STRUCT_H
#define STRUCT_H

typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;

#endif
```

Allocation dynamique de la mémoire (2/2)

- **Attention :**
  - Ne jamais accéder à la mémoire après l'avoir libérée
  - Ne jamais libérer plusieurs fois la mémoire

```
int *arr;
…
arr = (int*) malloc( n * sizeof(int) );
…
free( arr );
…
free( arr ) ;
```

```
int *arr;
…
arr = (int*) malloc( n * sizeof(int) );
…
free( arr ) ;
…
int j = arr[0];
```

  - Mais la mémoire doit être libérée
  - ➢ Sinon, fuite mémoire

Application à la gestion d'une liste chaînée (1/3)

```c
#include <stdlib.h>

typedef struct node {
    int data;
    struct node *next;
} Node ;

typedef struct {
    Node *first;
} List;

/* listCreate
 *  Creates an empty linked list.
 *
 * Returns:
 *  allocated list, holding nothing.
 */
List* listCreate() {
  List *ret;

  ret = (List*) malloc( sizeof(List) );
  ret->first = NULL;
  return ret;
}

…
```

Application à la gestion d'une liste chaînée (2/3)

```
/* listRemove
 *  Removes first occurrence of number from a
 *  linked list, returning 1 if successful
 *
 * Parameters:
 *  list - list from which item is to be removed.
 *  to_remove - number to be removed from the list.
 *
 * Returns:
 *  1 if item was found and removed, 0 otherwise.
 */
int listRemove(List *list, int to_remove) {
  Node * prev, cur ;

  prev = NULL;
  cur = list->first
  while (cur != NULL) {
   if (cur->data == to_remove) {
     if (prev == NULL) {
       list->first = cur->next;
     } else {
       prev->next = cur->next;
     }
     free(cur);
     return 1;
   }
   prev = cur;
   cur = cur->next;
 }
 return 0;
}
```

Application à la gestion d'une liste chaînée (3/3)

- Une autre implémentation possible de `listRemove` :

```
int listRemove(List *list, int to_remove) {
  Node **cur_p;
  Node *out;
  int cur_data;

  cur_p = &(list->first);
  while (*cur_p != NULL) {
    cur_data = (*cur_p)->data;
    if (cur_data == to_remove) {
      out = *cur_p;
      *cur_p = out->next;
      free(out);
      return 1;
    }
    cur_p = &((*cur_p)->next);
  }
  return 0;
}
```

- curr_p représente un pointeur vers le pointeur du premier nœud
  - i.e. l'adresse en mémoire où est stocké l'adresse du premier nœud

➢ Permet de changer ces adresses et donc le chaînage de la liste