Chaînes de caractères

G. Dewaele

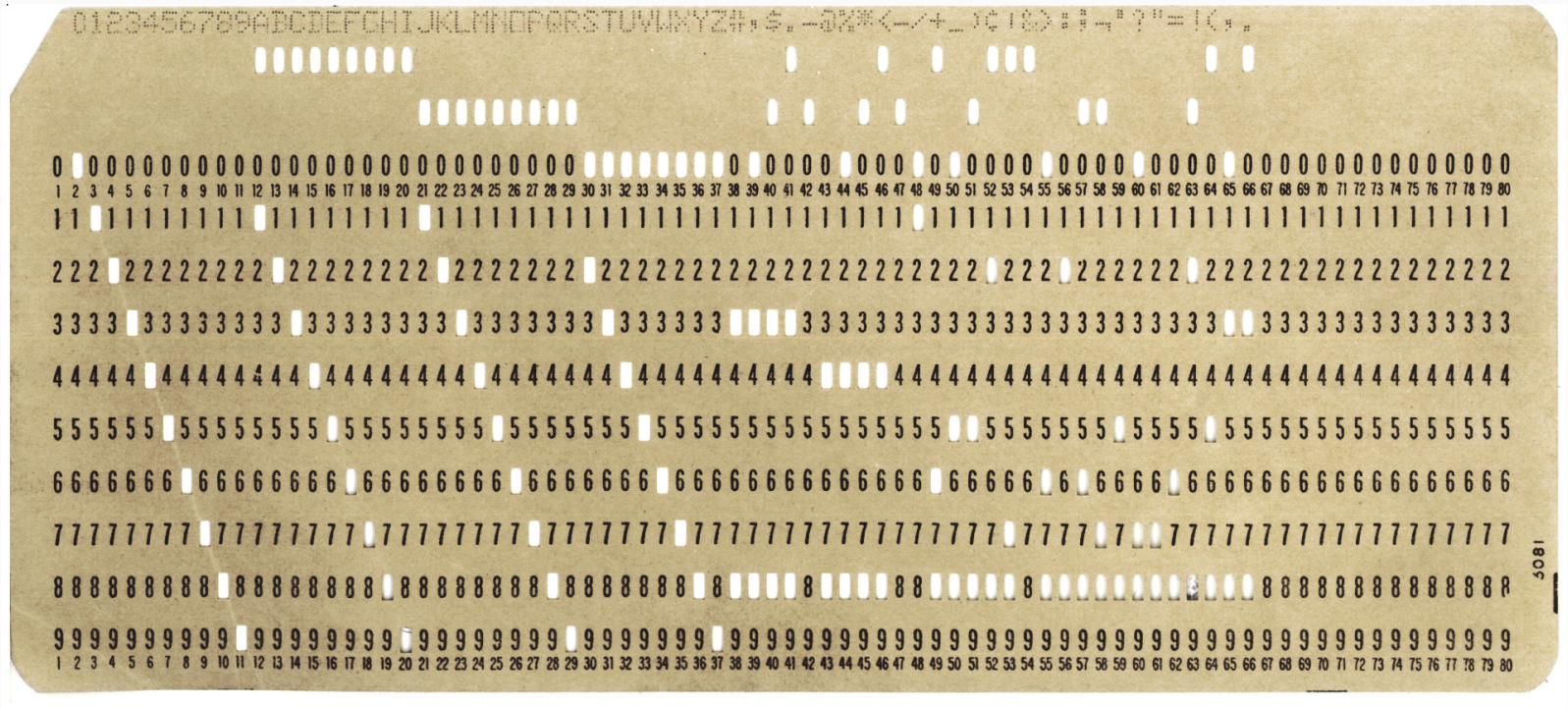
Le problème à résoudre

On veut également pouvoir ranger du texte en mémoire !

Idée simple : un caractère ≡ un entier naturel
« point de code »

Premiers pas

Chaque fabricant choisit son propre encodage



(lié à EBCDIC)

C'est le chaos...

Des premières ébauches de standardisation

Vers 1960 : American Standard Code for Information Interchange

	000...	001...	010...	011...	100...	101...	110...	111...
...0000	NULL	DLE		0	@	P	‘	p
...0001	SOH	DC1	!	1	A	Q	a	q
...0010	STX	DC2	”	2	B	R	b	r
...0011	ETX	DC3	#	3	C	S	c	s
...0100	EOT	DC4	$	4	D	T	d	t
...0101	ENQ	NAK	%	5	E	U	e	u
...0110	ACK	SYN	&	6	F	V	f	v
...0111	BEL	ETB	’	7	G	W	g	w
...1000	BS	CAN	(	8	H	X	h	x
...1001	TAB	EM	)	9	I	Y	i	y
...1010	LF	SUB	*	:	J	Z	j	z
...1011	VT	ESC	+	;	K	[	k	{
...1100	FF	FS	,	<	L	\	l	|
...1101	CR	GS	-	=	M	]	l	}
...1110	SO	RS	.	>	N	^	n	~
...1111	SI	US	/	?	O	_	o	DEL

Le code ASCII

Code sur 7 bits (dans $[\![0\,..\,127]\!]$)

(possibilité d'avoir un bit de contrôle)

Nombreux codes de contrôle

- pour les communications (SYN, ACK, NAK, EOT)
- pour le conditionnement des données (SOH, STX, ETX)
- pour la gestion de leur interprétation (SI, SO, DLE, ESC)
- pour le contrôle des périphériques (DC1... DC4)
- pour la gestion de la position (BS, TAB, VT, CR, LF, FF)...

Standardisation ISO/IEC 646

Standardisation dans la norme ISO/IEC 646

Plusieurs variantes pour d'autres langues que l'anglais

Un caractère typographique n'est pas forcément sur 7 bits

par exemple, « 60 08 65 » (≡ ` BS e) ⟼ è

Variantes locales

ISO646-IRV (≡ US)

	000…	001…	010…	011…	100…	101…	110…	111…
…0000	NULL	DLE		0	@	P	‘	p
…0001	SOH	DC1	!	1	A	Q	a	q
…0010	STX	DC2	”	2	B	R	b	r
…0011	ETX	DC3	#	3	C	S	c	s
…0100	EOT	DC4	$	4	D	T	d	t
…0101	ENQ	NAK	%	5	E	U	e	u
…0110	ACK	SYN	&	6	F	V	f	v
…0111	BEL	ETB	’	7	G	W	g	w
…1000	BS	CAN	(	8	H	X	h	x
…1001	TAB	EM	)	9	I	Y	i	y
…1010	LF	SUB	*	:	J	Z	j	z
…1011	VT	ESC	+	;	K	[	k	{
…1100	FF	FS	,	<	L	\	l	|
…1101	CR	GS	-	=	M	]	l	}
…1110	SO	RS	.	>	N	^	n	~
…1111	SI	US	/	?	O	_	o	DEL

Variantes locales

ISO646-FR et ISO646-FR-0

	000...	001...	010...	011...	100...	101...	110...	111...
...0000	NULL	DLE		0	à	P	µ	p
...0001	SOH	DC1	!	1	A	Q	a	q
...0010	STX	DC2	”	2	B	R	b	r
...0011	ETX	DC3	■	3	C	S	c	s
...0100	EOT	DC4	$	4	D	T	d	t
...0101	ENQ	NAK	%	5	E	U	e	u
...0110	ACK	SYN	&	6	F	V	f	v
...0111	BEL	ETB	’	7	G	W	g	w
...1000	BS	CAN	(	8	H	X	h	x
...1001	TAB	EM	)	9	I	Y	i	y
...1010	LF	SUB	*	:	J	Z	j	z
...1011	VT	ESC	+	;	K	°	k	é
...1100	FF	FS	,	<	L	ç	l	ù
...1101	CR	GS	-	=	M	§	l	è
...1110	SO	RS	.	>	N	^	n	¨
...1111	SI	US	/	?	O	_	o	DEL

Les extensions

On ne peut plus écrire de C !

Les extensions

On ne peut plus écrire de C !

{ et } peuvent être remplacés par ??< et ??>

Les extensions

On ne peut plus écrire de C !

{ et } peuvent être remplacés par ??< et ??>

Comme ce n'est pas très lisible, on a aussi <% et %>

Les extensions

On ne peut plus écrire de C !

{ et } peuvent être remplacés par ??< et ??>

Comme ce n'est pas très lisible, on a aussi <% et %>

```c
bool foo(bool t??(??), unsigned i, unsigned j) ??<
    if (t<:i:> ??!??! t<:j:> <% return i??'j; %>
    return ??-i;
??>
```

Bon, OK, ça reste illisible

Les extensions

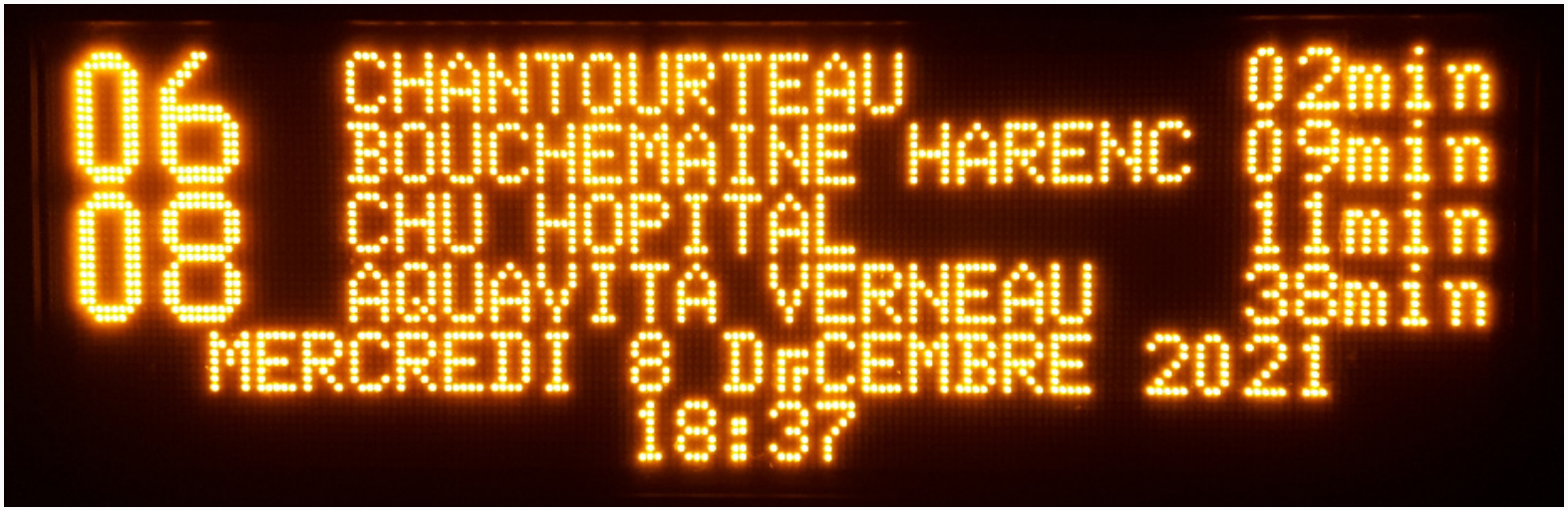
Idée : profiter des 128 codes restants !

À nouveau, quantité de standards

- IBM CP437, CP850, CP863...
- ISO/CEI 8859-1 (latin1) à 8859-16
- Microsoft CP1252
- ISO-8859-1
- ...

Les extensions

Le résultat :



	Latin9 (ISO 8859-15)	CP457
0xC9	É	╔
0x90	invalide	É

Le standard Unicode

Points de code entre `0x000000` et `0x10FFFF`

Plus de 144000 attribués (sur 1112064 possibles)

Dont symboles, emojis, codes de contrôle...

Les 256 premiers basés sur ISO 8859-1 (Latin1)

Le standard Unicode

On ne veut pas utiliser 21 bits...

Traduction en séquences d'octets de longueur variable

- UTF-8
- UTF-16
- UCS-2
- UTF-32...

Les chaînes de caractères

Il faut donc distinguer

- une séquence de caractères/symboles
- une succession de bytes encodant cette séquence

Certains langages permettent de manipuler les symboles

Les chaînes de caractères en Python 3

```
In [1]: ch = "J'❤ OCaml et C"

In [2]: len(ch)
Out[2]: 14

In [3]: code = ch.encode("utf8")

In [4]: " ".join(hex(c)[2:] for c in code)
Out[4]: '4a 27 f0 9f a7 a1 20 4f 43 61 6d 6c 20 65 74 20 43'

In [5]: len(code)
Out[5]: 17

In [6]: code.decode("cp1252")
Out[6]: "J'ðŸ§¡ OCaml et C"
```

Difficultés

Beaucoup de difficultés pratiques :

- que doit-on considérer comme « élémentaire » ?
- plusieurs représentations pour une même chaîne...

La longueur, l'égalité posent problème

Le stockage également !

Les chaînes de caractères en C et OCaml

On ne considère que des tableaux de `bytes`

Les bibliothèques standard n'interprètent pas les codes

Module OCaml `String` : « A string s of length n is an indexable and immutable sequence of n bytes. For historical reasons these bytes are referred to as characters »

Les chaînes de caractères en C

On utilise des tableaux de **char**

Les fonctions manipulant des chaînes utilisent des **char***

Les chaînes de caractères en C

Problème : où se termine la chaîne ?

Solution : on utilise une *sentinelle*, le code `0x00`

« `Hello MP2I!` »

↦ « `48 65 6c 6c 6f 20 4d 50 32 49 21 00` »

Littéraux

Pour définir une chaîne :

```c
char ch[] = {0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20,
             0x4d, 0x50, 0x32, 0x49, 0x21, 0x00};
```

```c
char ch[] = {'H', 'e', 'l', 'l', 'o', ' ',
             'M', 'P', '2', 'I', '!', '\0'};
```

Ou, plus simple :

```c
char ch[] = "Hello MP2I!";
```

Manipuler une chaîne de caractères

Une chaîne se manipule comme un tableau

ch[i] désigne le **char** d'index i

Si on écrit :

```c
char ch[] = "Hello MP2I!";


ch[5] = 21;
ch[6] = 0;
```

Un affichage de ch donne « Hello! »

(taille en mémoire inchangée, &ch[7] désigne encore P2I!)

Manipuler une chaîne de caractères

On peut utiliser

```c
ch[5] = '!';
```

plutôt que

```c
ch[5] = 21;
```

Réserver de la place pour une chaîne

Pour réserver de la place en mémoire pour une chaîne :

```c
char ch[100] = { 0 };
```

Ou bien

```c
char* ch = malloc(100 * sizeof(char));
ch[0] = 0;
```

Il est prudent d'avoir toujours un 0 dans le tableau !

S'affranchir de l'encodage

Les codes binaires dans le fichier source

S'affranchir de l'encodage

Les codes binaires dans le fichier source

se retrouvent à l'identique dans l'exécutable binaire

S'affranchir de l'encodage

Les codes binaires dans le fichier source

se retrouvent à l'identique dans l'exécutable binaire

et sont reproduits tels quels en sortie

S'affranchir de l'encodage

Les codes binaires dans le fichier source

se retrouvent à l'identique dans l'exécutable binaire

et sont reproduits tels quels en sortie

Seuls " et \ nécessitent une interprétation particulière

Codes d'échappement

Des séquences spéciales pour les éléments problématiques :

« \" » pour représenter « " »

« \\ » pour représenter « \ »

« \0 » pour représenter le caractère de code 0x0

« \n » pour représenter un retour à la ligne

etc.

Retour à la ligne

Retour à la ligne

Différentes conventions pour les fins de ligne :

- Windows, Symbian OS : CRLF (`\r\n` soit `0x0d 0x0a`)
- Unix, OS-X : LF (`\n` soit `0x0a`)
- Mac OS : CR (`\r` soit `0x0d`)

Cause régulièrement de mauvaises surprises

Immutabilité des chaînes littérales

On peut aussi écrire

```c
char* ch = "Hello MP2I!";
```


MAIS les chaînes littérales produisent un tableau de **char**

localisé dans une zone mémoire immutable

Un **char** ch[] déclare un tableau dans une zone mutable

Immutabilité des chaînes littérales

On a donc

```c
char ch[] = "Hello MP2I!";

char c = ch[0];  // Correct
ch[0] = 'Z';     // Correct
```

En revanche :

```c
char* ch = "Hello MP2I!";

char c = ch[0];  // Correct
ch[0] = 'Z';     // Erreur de segmentation !
```

Quelques fonctions

La bibliothèque standard fournit des outils

Elle travaille avec des `char*`

Import des fonctions avec

```c
#include <string>
```

Longueur d'une chaîne

```
int strlen(char* str)
```

Retourne la longueur de `str`

La chaîne doit se terminer par `\0`

Le `\0` n'est pas compté

Plus sûr :

```
strnlen_s(char* str, size_t strsz)
```

Comparer des chaînes

Cela peut s'écrire :

```c
int my_strlen(char* str) {
    int i=0;

    while (str[i] != 0) { i++; }

    return i;
}
```

Comparer des chaînes

```
int strcmp(char* lhs, char* rhs)
```

Retourne `0` si égales, `-1` si `lhs` $\preceq_{\mathrm{L}}$ `rhs`, `+1` sinon

Les chaînes doivent se terminer par `\0`

Les caractères sont comparés comme **`unsigned char`**

Plus sûr :

```
int strncmp(char* lhs, char* rhs, size_t count)
```

Comparer des chaînes


```c
int my_strcmp(char* lhs, char* rhs) {
    int i=0;
    while (lhs[i] != 0 && lhs[i] == rhs[i]) { i++; }

    if (lhs[i] == 0 && rhs[i] == 0) { return 0; }
    if (lhs[i] == 0) { return -1; }
    if (rhs[i] == 0) { return 1; }

    unsigned char cl = lhs[i], cr = rhs[i];
    if (cl < cr) { return -1; } else { return 1; }
}
```

Copier des chaînes

```
char* strcpy(char* dest, char* src)
```

- Copie le contenu à l'adresse `src` vers l'adresse `dest`
- Les chaînes doivent se terminer par `\0`
- La destination doit être assez grande
- Les zones mémoires doivent être distinctes

Plus sûr : `strncpy`, `strcpy_s`, `strncpy_s`...

Les risques

Considérons par exemple :

```c
char ch1[] = "Hello";
char c = 37;
char ch2[] = "World!";

strcpy(ch1, "Hello * MP2I!");
```

Si c et ch2 se trouvent après en mémoire

avec `strcpy`, ils sont écrasés !

Entrées de l'utilisateur

On peut utiliser scanf pour lire des données

Pour lire un entier :

```c
int val = 0;
scanf("%d", &val);
```

Pour une chaîne de caractères :

```c
char buffer[100] = { 0 };
scanf("%s", buffer);
```

Super, merci !


```c
char invite[] = "Hello";
char value = 37;
bool locked = true;
char name[5] = { 0 };

printf("Entrez votre nom : ");
scanf("%s", name);

printf("\n%s %s!\n\n", invite, name);

if (locked) { foo_locked(); }
    else { foo_unlocked(); }
```

Super, merci !

Bilan :

- tout débordement de buffer peut avoir des conséquences

Super, merci !

Bilan :

- tout débordement de buffer peut avoir des conséquences
- scanf peut être très dangereux

Super, merci !

Bilan :

- tout débordement de buffer peut avoir des conséquences
- scanf peut être très dangereux
- la MP2I ouvre des portes

Super, merci !

Bilan :

- tout débordement de buffer peut avoir des conséquences
- scanf peut être très dangereux
- la MP2I ouvre des portes

Solutions possibles :

- `scanf("%4s", buffer);` (attention au `\0`)
- `fgets(buffer, 5, stdin);`

Copier des données binaires

```
void* memcpy(void* dest, void* src, size_t count)
```

Copie count bytes de src vers dest

Ignore les éventuels \0

```
void* memset(void* dest, int ch, size_t count)
```

Copie count fois ch converti en **unsigned char** vers dest

Quelques autres fonctions

```
char* strchr(char* str, int ch)
```

première occurrence de ch

```
char* strrchr(char* str, int ch)
```

dernière occurrence de ch

```
char* strstr(char* str, char* substr)
```

première occurrence de substr

```
char* strcat(char* dest, char *src)
```

ajout de src à la fin de dest

Quelques autres fonctions

```
int atoi(char* str)
```

Lecture d'un entier

```
double atof(char* str)
```

Lecture d'un flottant

Recherche de sous-chaîne


```c
char* my_strstr(char* str, char* substr) {
    int n=strlen(str), ns=strlen(substr);

    for (int i=0; i<n-ns+1; ++i) {
        int j = 0;
        while (j<ns && str[i+j]==substr[j]) {
            j++;
        }
        if (j == ns) {
            return &(str[i]);
        }
    }
    return NULL;
}
```

Recherche de sous-chaîne

Complexité en $O((n - n_s + 1) \times n_s)$ On fera bien mieux !

Cette complexité est bien atteinte :

Exemple : recherche de aaaaab dans aaaaaaaaaaaaa...

Désigner un caractère dans le source

On peut désigner un « caractère » en écrivant « `'x'` »

Cela définit en fait un... `int` !

Il faut qu'il n'y ait qu'un seul byte dans le source entre les `'`

« `(c == 'à')` » peut être imprévisible

Et en OCaml ?

Un type **`char`**

littéraux avec `'` (ex : `'a'`, `'\n'`, `'\000'`...)

Un type **`string`**

littéraux avec `"` (ex : `"Hello MP2I!"`)

≡ suite d'octets, mais représentation interne non spécifiée

les chaînes sont immutables (il existe un type `bytes` mutable)

la taille est mémorisée en interne, `\000` peut se trouver dans la chaîne

Accéder à un caractère


```
# let ch = "Hello MP2I!";;
val ch : string = "Hello MP2I!"

# String.get;;
- : string -> int -> char = <fun>

# String.get ch 6;;
- : char = 'M'

# ch.[6];;
- : char = 'M'
```

Quelques fonctions

Pour obtenir la longueur d'une chaîne :

```
# String.length;;
- : string -> int = <fun>
```


Le calcul de la taille est en temps constant $O(1)$

Quelques fonctions

Pour obtenir une sous-chaîne :

```
# String.sub;;
- : string -> int -> int -> string = <fun>
```


Le second entier est la longueur du résultat

Crée une nouvelle chaîne en la copiant

(Complexité liée à la taille du résultat)

Quelques fonctions

Pour créer une chaîne répétition d'un même caractère :

```
# String.make;;
- : int -> char -> string = <fun>
```


Pour obtenir la première occurrence d'un caractère :

```
# String.index;;
- : string -> char -> int = <fun>
```