



Chapitre 8 : Chaînes de caractères

Sommaire

1. Le type char
 - ASCII
 - <cctype>
2. Chaînes littérales
3. La classe std::string
4. Opérateurs et méthodes de std::string
5. std::string_view
6. Au-delà d'ASCII (pour info)



1. Le type char



- Le type `char` permet de stocker des **entiers** stockés sur **1 Byte** et peut être précédé d'un **modificateur** ou non
 - `signed char` pour des entiers entre **-128 et 127**
 - `unsigned char` pour des entiers entre **0 et 255**
 - `char` soit signé soit non signé selon le compilateur
- Il permet surtout de stocker des **caractères** via un **codage** qui fait correspondre caractères et valeurs numériques → **ASCII**
- Le modificateur de signe **n'affecte pas le caractère** codé par les 8 bits du `char`
- Les **constantes littérales** de type `char` s'écrivent entourées **d'apostrophes**
 - `'a'` pour le `a` minuscule
 - `'3'` pour le caractère `3` (différent de la valeur entière `3`)



Codes de contrôle
Chiffres
Lettres majuscules
Lettres minuscules
Caractères spéciaux

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	í	¢	£	¤	¥	¡	§	”	©	¤	«	¬	□	®	-
Bx	°	±	²	³	‘	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ö	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ



- Le type transmis au flux par l'opérateur de flux (`<<`) détermine si on affiche le caractère ou la valeur numérique correspondante

```
cout << 'A' << " " << (int)'A' << endl;  
  
cout << (char)0x41 << " " << 0x41 << endl;  
  
cout << 'D' + 'a' - 'A' << " "  
    << char('D' + 'a' - 'A') << endl;
```

A	65
A	65
100	d

- Attention aux promotions de type !

Quelques constantes particulières



\n	passage à la ligne
\r	retour chariot « CR »
\t	tabulation horizontale
\v	tabulation verticale
\a	alerte sonore
\\"	barre oblique inverse « \ »
\'	caractère apostrophe « ' »
\"	caractère guillemet « " »
\b	retour arrière « backspace »
\f	saut de page « form feed »



- La librairie `<cctype>` met à disposition des **fonctions** qui permettent de savoir à quelle **catégorie** appartient un caractère

```
int isxxx(int c);
```

- Ces fonctions retournent un entier (pour des raisons historiques)
 - non nul (`true`) si le caractère `c` est `xxx`
 - et zéro (`false`) sinon

<http://www.cplusplus.com/reference/cctype>

Caractères – catégories



```
int isalnum(int c);
```

est une lettre de l'alphabet ou un chiffre

```
int isalpha(int c);
```

est une lettre de l'alphabet (minuscule ou majuscule)

```
int iscntrl(int c);
```

est un caractère de contrôle (code 0 à 31 et 127 (DEL))

```
int isdigit(int c);
```

est un chiffre

```
int isxdigit(int c);
```

est un chiffre hexadécimal

```
int isgraph(int c);
```

est affichable et non blanc (code 33 à 126)



```
int islower(int c);
```

est une lettre de l'alphabet minuscule

```
int isupper(int c);
```

est une lettre de l'alphabet majuscule

```
int isprint(int c);
```

est un caractère affichable
(code 32 à 126)

```
int ispunct(int c);
```

est un caractère de ponctuation

```
int isspace(int c);
```

est un espace, un tab, une fin de ligne
ou un retour

```
int isblank(int c);
```

est un espace ou un tab



- <cctype> fournit également deux fonctions de transformation

```
int tolower(int c);
```

```
int toupper(int c);
```

qui retournent respectivement la **minuscule** et la **majuscule** du caractère transmis

- Si la transformation n'est **pas possible** (caractères non alphabétiques) elles retournent le **caractère original inchangé**

2. Chaînes littérales



Quand il y a plus d'un caractère...



- Les chaînes de caractères littérales s'écrivent entourées de **doubles guillemets**
- Pour écrire le caractère " au milieu d'une chaîne, on le précède d'un backslash. Pour écrire un backslash, on le précède également d'un backslash
- On peut écrire une **chaîne sur plusieurs lignes** en finissant chaque ligne par \
- Il est possible d'écrire une chaîne en plusieurs parties qui se suivent

```
int main() {
    cout << "Hello, World !" << endl;
    cout << " \" et \\ au milieu d'une
chaine"
        << endl;
    cout << "Une chaine \
ecrite sur plusieurs \
lignes" << endl;
    cout << "Une"
        " chaine "
        "en " "plusieurs"
        "parties" << endl;
}
```

```
Hello, World !
" et \ au milieu d'une chaine
Une chaine ecrite sur plusieurs lignes
Une chaine en plusieursparties
```



- Le préfixe R permet de créer des chaines de caractères brutes, où les caractères spéciaux tels que " ou \ ne sont pas interprétés
- La chaine brute est entourée de R"(et ") ou R"xyz(et "xyz) avec xyz quelconque

```
// syntaxe originale
cout << " \" et \\ au milieu d'une chaine" << endl;

// syntaxe chaine brute
cout << R"( \" et \\ au milieu d'une chaine)" << endl;

// syntaxe chaine brute alternative
cout << R"abc(Placer les memes caracteres entre les \"( initial et )" final
permet d'ecrire )" dans la chaine brute )abc" << endl;
```

```
" et \" au milieu d'une chaine
" et \\ au milieu d'une chaine
Placer les memes caracteres entre les \"( initial et )" final permet
d'ecrire )" dans la chaine brute
```



- Notons que ces constantes littérales sont de type `const char*` et non du type `string`. Ce sont des tableaux de `char`, terminés conventionnellement par le caractère zéro. Le traitement de ce type de chaînes sera étudié en PRG2.
- En pratique, on peut presque toujours ignorer cela, vu qu'il y aura **conversion implicite** vers le type `string` quand c'est nécessaire
- Il est également possible d'effectuer une **conversion explicite** ou de spécifier le type `string` en ajoutant la lettre s en **suffixe** de la chaîne littérale. `s1` et `s2` ci-dessous sont de type `string`

```
auto s1 = string("hello");    // conversion explicite
auto s2 = "hello"s;          // conversion par suffixe
```

3. Classe std::string





- On peut stocker une chaîne de caractères dans une variable de type `std::string`
- Ce n'est pas un type fondamental du C++. Il est défini dans la librairie `<string>`, qu'il faut inclure

```
#include <string>
using namespace std;

...
string s = "Hello, World!";
```



- Contrairement aux types simples une variable non initialisée **est définie**

```
string s0; // contient une chaîne vide
```

- On peut initialiser les variables de type **string** des trois manières habituelles
 - avec une **chaîne littérale** en paramètre

```
string s1 = "Hello, World!";
string s2("Hello, World!");
string s3{"Hello, World!"};
```

- ou avec un autre `std::string` en paramètre qui est **copié**

```
string s4 = s1;
string s5(s1);
string s6{s1};
```



- Il est également possible d'initialiser un `std::string` avec plusieurs paramètres. On utilise alors la syntaxe de constructeur (avec parenthèses)
- Le constructeur de remplissage `string(size_t n, char c)` crée une chaîne contenant n fois le caractère c

```
string s1(6, 'a'); // "aaaaaa"
```

- Le constructeur depuis un buffer `string(const char* p, size_t n)` crée une chaîne contenant les n premiers caractères du buffer commençant à l'adresse p

```
const char* p = "Hello, world !";
string s2(p,5);                      // "hello"
string s3("Hello, World!", 4);        // "hell"
string s4(s2.data(),2);               // "he"
```



- Le constructeur `string(string str, size_t pos, size_t len)` crée une chaîne de `len` caractères copiés à partir du caractère en position `pos` de la chaîne `str`
 - si `len` n'est pas précisé, `str` est copiée jusqu'à sa fin
 - on numérote les positions à partir de 0

```
// indices      01234567890123
string hello("Hello, World!");

string s1(hello, 0, 4);          // contient la chaîne "Hell"
string s2(hello, 7, 5);          // contient la chaîne "World"
string s3(hello, 7);             // contient la chaîne "World!"
string s4(hello);               // contient la chaîne "Hello, World!"
```

Attention !



Lorsque l'on construit une chaîne avec 2 paramètres dont le second est un entier, il ne faut pas confondre

- Le constructeur depuis un buffer, si le premier paramètre est une chaîne littérale
- Le constructeur par sous-chaîne avec le troisième paramètre `len` non spécifié, si le premier paramètre est de type `std::string`

```
string hello("Hello, World!");

string s1("Hello, World!", 7);           // buffer : contient "Hello, "

string s2(hello, 7);                   // substr : contient "World!"
string s3(string("Hello, World!"), 7); // substr : contient "World!"
```

4. Opérateurs et méthodes



René Descartes



- L'opérateur `=` permet d'affecter une nouvelle valeur à une string
- Il convertit implicitement des expressions de type char ou des chaînes comme en C, y compris les constantes littérales

```
string str1, str2, str3;
str1 = "Test string: " // chaîne C littérale
str2 = 'x';           // caractère
str3 = str1;          // string
```



La méthode `.assign(...)` offre plus d'options que l'opérateur. Elle s'invoque avec les mêmes arguments que les constructeurs

```
string str,  
      world("World!"),  
      hw("Hello, World!");  
  
// Les lignes suivantes sont équivalentes  
str.assign(world);  
str.assign(hw, 7, 6);  
str.assign("World!");  
str.assign("World!!!!", 6);  
  
// cette ligne donne la chaîne "WWWWWWWW"  
str.assign(6, 'W');
```

Concaténation – l'opérateur +



- L'opérateur **+** permet de concaténer deux chaînes.
 - Un des 2 paramètres peut être une chaîne littérale
 - Ou un caractère seul
- Par contre, on ne peut pas l'utiliser pour concaténer
 - deux chaines littérales
 - un std::string et un entier
 - Une chaine littérale et un char

```
string hello("Hello, " );
string world("World!");
string s1 = hello + world;      // "Hello, World!"
```

```
string s2 = "Hello, " + world; // "Hello, World!"
string s3 = hello + "World!"; // "Hello, World!"
```

```
string s4 = hello + 'W';      // "Hello, W"
string s5 = 'W' + hello;       // "WHello, "
```

```
string erreur1 = "Hello, " + "World!";
// ne compile pas
string erreur2 = hello + 5;
// ne compile pas
string erreur3 = "Hello, " + 'W';
// comportement indéfini
```



Concaténation – l'opérateur `+=`



Comme pour les opérateurs sur les entiers et les réels, il y a un **opérateur auto-affecté** correspondant, qui accepte les `char`, les `string` et les chaînes littérales

```
string str("Hello");

str += ',';           // même effet que str = str + ',';
                     // str contient "Hello,"

str += " World";    // même effet que str = str + " World";
                     // str contient maintenant "Hello, World"

string exclamation("!");
str += exclamation; // même effet que str = str + exclamation;
                     // str contient maintenant "Hello, World!"
```

Concaténation – la méthode append



La méthode `.append(...)` offre plus d'options que les opérateurs. Elle s'invoque avec les mêmes arguments que les constructeurs

```
string str("Hello, "),  
       world("World!"),  
       hw("Hello, World!");  
  
// Les lignes suivantes sont équivalentes  
str.append(world);  
str.append(hw, 7, 6);  
str.append("World!");  
str.append("World!!!!", 6);  
  
// La ligne suivante ajoute la chaîne "WWWWWW"  
str.append(6, 'W');
```



Pour une chaîne `str` et un entier `i`, l'expression `str[i]` permet d'accéder en lecture comme en écriture au `i`^{ème} caractère – en numérotant depuis 0.

```
string hello("Hello, World!");
char fifth = hello[4];
hello[4] = ' ';

cout << hello << endl;
cout << fifth << " remplacé par un blanc" << endl;
```

```
Hello, World!
o remplacé par un blanc
```



- L'opérateur `[]` a un comportement indéfini si on lui donne un paramètre hors de l'intervalle compris entre 0 et la longueur de la chaîne moins 1
- La méthode `.at(i)` permet un accès identique mais plus sécurisé au *i*^{ème} caractère **en vérifiant que i est dans le bon intervalle.** Elle lance une exception spécifique sinon (c.f. chap 14)

```
string hello("Hello, World!");
char fifth = hello.at(4);
hello.at(4) = ' ';

cout << hello << endl;
cout << fifth << " remplacé par un blanc" << endl;
```



- On peut demander la **longueur** d'une chaîne avec les méthodes `.length()` ou `.size()`
- La valeur rentrée est de type `size_t`, un type entier non signé dont C++ garantit qu'il est assez grand pour stocker la taille de toute chaîne, tableau, objet, ...

```
string s("Hello");
size_t longueur = s.length(); // 5
size_t taille   = s.size();   // 5 aussi
```

- La méthode `.empty()` indique une longueur nulle.

```
string s("Hello");

bool vide1 = s.size() == 0; // false
bool vide2 = s.empty();    // idem, mais plus clair
```

Modification de la taille



La méthode `.resize(size_t len, char c);` modifie la longueur de la chaîne

- `len` spécifie la nouvelle taille
- `c` spécifie le caractère utilisé pour compléter la chaîne si sa taille augmente. Le caractère nul '`\0`' est utilisé par défaut

```
// 01234567
string s("Hello");      // "Hello"
s.resize(8, '!');        // "Hello!!!"
s.resize(4);              // "HeLL"
s.resize(6);              // "HeLL\0\0"
```



La méthode `.substr(size_t pos, size_t len);` permet d'extraire une sous-chaîne de `len` caractères en commençant à la position `pos`

- si `len` manque, on extrait jusqu'à la fin
- si `pos` manque, on extrait depuis le début

```
//      0123456789*12
string s("Hello, World!");
string s1 = s.substr(0, 5);      // "Hello"
string s2 = s.substr(7, 5);      // "World"
string s3 = s.substr(7);        // "World!"
string s4 = s.substr();         // "Hello, World!"
```



La méthode `.insert(size_t pos, string str)` insère la chaîne `str` en position `pos`

- Augmente la taille de la chaîne sauf si `str` est ""
- `str` peut être remplacé comme pour les méthodes `append`, `assign`, ...

```
string s("to be question"), s2("the "), s3("for not to be");

s.insert(6, s2);                      // to be (the )question
s.insert(6, s3, 1, 7);                // to be (or not )the question
s.insert(13, "that is cool", 8);     // to be or not (that is )the question
s.insert(13, "to be ");               // to be or not (to be )that is the question
s.insert(18, 1, ':' );                // to be or not to be(:) that is the question
```



`.replace(size_t pos, size_t len, string str)` remplace par la chaîne `str` la sous-chaîne de longueur `len` débutant en position `pos`

- modifie la taille de la chaîne si `str.length()` est différente de `len`
- `str` peut être remplacé comme pour la méthode `append`

```
string s1 = "n example";
string s2 = "sample phrase";
string s =
    "this is a test string.";
    // 0123456789*123456789*12345
s.replace(9, 5, s1);
    // "this is an example string."
s.replace(19, 6, s2, 7, 6);
    // "this is an example phrase."
s.replace(8, 10, "just a");
    // "this is just a phrase."
s.replace(8, 6, "a shorty", 7);
    // "this is a short phrase."
s.replace(22, 1, 3, '!');
    // "this is a short phrase!!!"
```



- La méthode `.clear();` vide la chaîne
- La méthode `.erase(size_t pos, size_t len);` efface `len` caractères à partir de la position `pos`
 - Jusqu'à la fin si `len` non spécifié
 - Depuis le début si `pos` non spécifié (synonyme de `clear()`)

```
string s("This is an example sentence.");
s.erase(9, 9); // "This is a sentence."
s.erase(13);   // "This is a sen"
s.erase();     // ""
```



- Les opérateurs d'affectation et les méthodes `assign`, `append`, `insert`, `replace` et `erase` ont comme **valeur de retour** une référence vers **la chaîne elle-même**
- Cela permet **d'enchaîner ces opérations**

```
string s = "Bonjour, le monde!";
cout << s.assign("hello").append(", Wworld!").replace(0,1,"H").erase(8,1);
```

Hello, World!



- La **recherche** dans une chaîne est une des opérations les plus courantes, pour laquelle on dispose de 6 méthodes

find	Trouve une sous-chaîne
rfind	Idem depuis la fin
find_first_of	Trouve le premier caractère parmi une liste
find_last_of	Idem depuis la fin
find_first_not_of	Trouve le premier caractère hors d'une liste
find_last_not_of	Idem depuis la fin

- Toutes ces méthodes renvoient la **position** trouvée (de type `size_t`)
- Si la recherche est **infructueuse** elles renvoient `string::npos` (= -1, donc `numeric_limits<size_t>::max()`)



find (rfind)

`.find(string str, size_t pos);` (`rfind`) trouve la première (dernière) occurrence de la sous-chaîne `str`, en cherchant depuis la position `pos`

- `str` peut être remplacé par un `char`, une constante littérale (`const char*`) ou par ses premiers caractères
- `pos` vaut zéro si elle n'est pas spécifiée

```
//          0123456789*123456789*123456789*123456789*123456789*1
string s("There are two needles in this haystack with needles.");
string s2("needle");

size_t needle1    = s.find(s2);
size_t needle2    = s.find(s2, needle1 + 1);
size_t haystack   = s.find("haystack");
needle1           = s.find("needles are small", 0, 6);
needle2           = s.find("needles are small", needle1 + 1, 6);
size_t point      = s.find('.');

s.replace(s.find(s2), s2.length(), "preposition");
cout << s << endl;
```

There are two prepositions in this haystack with needles.



find_first_of (find_last_of)

`.find_first_of(string str, size_t pos = 0);` (`find_last_of`) trouve la première (dernière) occurrence d'un caractère de la chaîne str (ou remplacements habituels), en cherchant depuis la position pos

```
// positions:      0123456789*1
string s = string("Hello World!");
string vowels = string("aeiouyAEIOUY"); // voyelles

cout << s.find_first_of(vowels) << endl;
cout << s.find_first_of(vowels, 5) << endl;
cout << s.find_first_of("Good Bye!") << endl;
cout << s.find_first_of("Good Bye!", 0, 4) << endl;
cout << s.find_first_of('x') << endl;
```

```
1
7
1
4
18446744073709551615
```



find_first_not_of (find_last_not_of)

`.find_first_not_of(string str, size_t pos = 0) (find_last_not_of)`
trouve la première (dernière) occurrence d'un caractère hors de la chaîne str (ou remplacements habituels), en cherchant depuis la position pos

```
// 0123456789*123456789*123456789*123456789*1234
string str("recherche de caracteres non-alphabetiques ...");

size_t found = str.find_first_not_of("abcdefghijklmnopqrstuvwxyz ");

cout << "Le premier caractere non-alphabetique est '"
    << str.at(found)
    << »' a la position "
    << found << endl;
```

```
Le premier caractere non-alphabetique est '-' a la position 27
```

5. string_view





- Considérons le code suivant

```
void imprimer(std::string str){  
    std::cout << str << '\n';  
}  
  
int main() {  
    std::string s{"Hello, world!"};  
    imprimer(s);  
}
```

- La chaîne "Hello, world!" est copiée deux fois
 - Lors de l'initialisation de s
 - Lors du passage du paramètre de `imprimer` par valeur

Enoncé du problème



- En passant le paramètre par `const&`, on évite une copie. C'est mieux ...

```
void imprimer(const std::string& str){  
    std::cout << str << '\n';  
}
```

- Mais ce n'est pas satisfaisant avec le code appelant suivant

```
int main() {  
    imprimer("Hello, world!");  
}
```

- Pour pouvoir appeler `imprimer`, la chaîne littérale est convertie en un `std::string` temporaire, ce qui implique une copie ...



- Pour éviter toute copie dans ces cas, C++17 a introduit le type `std::string_view` dans l'en-tête `<string_view>` qui permet d'accéder **en lecture seulement** à une chaîne de caractères stockée « ailleurs ».
- C'est une structure légère qui stocke
 - Un `const char*` vers le début des caractères vus
 - Un `size_t` pour le nombre de caractères vus
 - Mais **pas les caractères** eux-mêmes, qui doivent résider dans un autre objet : `std::string`, chaîne C, chaîne littérale,

Exemples

- On peut initialiser `string_view` avec **deux paramètres** : une adresse et un nombre de caractères.
 - Une chaîne littérale est de type `const char*`
 - L'adresse du premier caractère de la chaîne s'est `&s[0]`. C'est aussi l'adresse rentrée par la méthode `s.data()`
- On peut aussi l'initialiser avec **un seul paramètre** pour voir toute la chaîne

```
#include <iostream>
#include <string>
#include <string_view>

using namespace std;

int main() {
    auto chaine_litterale = "Hello, world!";
    cout << string_view{chaine_litterale, 13} << endl;
    cout << string_view(chaine_litterale, 5) << endl;
    cout << string_view{chaine_litterale + 7, 5} << endl;

    string chaine_cpp = "Hello, world!";
    cout << string_view(&chaine_cpp[0], 13) << endl;
    cout << string_view(chaine_cpp.data(), 13) << endl;
    cout << string_view(chaine_cpp.data(), 5) << endl;
    cout << string_view(&chaine_cpp[7], 5) << endl;

    cout << string_view("Hello, world!") << endl;
    cout << string_view(chaine_litterale) << endl;
    cout << string_view(chaine_cpp) << endl;
}
```

```
Hello, world!
Hello
World
Hello, world!
Hello, world!
Hello
world
Hello, world!
Hello, world!
Hello, world!
```

std::string_view en paramètre



- En choisissant le type `std::string_view` passé par valeur en paramètre, aucun des trois appels ci-dessous ne copie de caractères.
- Pour un accès en lecture seule aux caractères, c'est la méthode à privilégier.

```
#include <iostream>
#include <string>
#include <string_view>

void imprimer(std::string_view str){
    std::cout << str << '\n';
}

using namespace std;

int main() {
    auto chaine_litterale = "Hello, world!";
    std::string chaine_cpp = "Hello, world!";

    imprimer("Hello, world!");
    imprimer(chaine_litterale);
    imprimer(chaine_cpp);
}
```



- `std::string_view` présente une interface similaire à celle de `const std::string&`, i.e. une interface **restreinte aux fonctions membres qui ne modifient pas** la chaîne.
 - Elle ne dispose pas de `.assign`, `append`, `insert`, `erase`, `replace`, `resize`, `clear`, `push_back`, et `pop_back`
 - `operator[]` et `.at()` retournent un `const char&`, pas un `char&`
- `substr` retourne une `string_view`, pas une `string`

```
string_view sv1{"Hello, world!"};
string_view sv2 = sv1.substr(0, 5); // "Hello"
string_view sv3{"world"};
size_t pos = sv1.find(sv3);           // pos = 7

// sv1.append(" PRG1");
// Compilation error: no member named 'append' in 'std::string_view'
```



- L'initialisation d'une `string_view` à partir d'une `string` fonctionne avec les 3 syntaxes.
- L'affectation dans une `string_view` change quelle chaîne est vue
- L'initialisation d'une `string` à partir d'une `string_view` requiert une **conversion explicite**. Seules les syntaxes avec () ou {} le permettent

```
string s = "Hello";
string_view sv1{s}; // sv1 voit s
string_view sv2(s); // sv2 voit s
string_view sv3 = s; // sv3 voit s
sv1 = "World"; // sv1 voit maintenant "World"

string s1(sv1); // s1 est une copie de "World"
string s2{sv2}; // s2 est une copie de "Hello"

// string s3 = sv2;
// Compilation error: no viable conversion from
// 'std::string_view' to 'std::string'

string s4 = string(sv2); // conversion explicite.
// s4 est une copie de "Hello"
```

6. Au-delà du codage ASCII

(pour information)





- A la création du langage C, on utilise le code **ASCII**, stocké sur **7 bits**
 - ne code pas les lettres accentuées
 - codes 0x00 à 0x1F et 0x7F pour le contrôle (caractères non affichables)

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL



- La plupart des ordinateurs stockent les informations par blocs de 8 bits
Que faire du 8ème bit ? Ce que l'on veut...
 - WordStar – un traitement de texte – l'utilise pour indiquer qu'une lettre termine un mot
 - L'ensemble de **caractères OEM** l'utilise pour fournir quelques lettres accentuées, mais aussi des caractères de dessin de lignes
 - Hors des USA, d'autres **pages de code** sont utilisés pour coder les lettres **d'autres langues**. Par exemple, 130 code é aux USA, mais Gimel נ en Israël

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Ø	ø	♥	♦	♣	♠					δ	♀	∏	*		
1	►	◀	↑	↓	¶	§	-	‡	↑	↓	→	←	↔	▲	▼	
2	!	"	#	\$	%	&	,	<	>	*	+	,	=	:	/	
3	Ø	ø	1	2	3	4	5	6	7	8	9	:	;	<	=	?
4	€	¤	A	B	C	D	E	F	G	H	I	J	K	L	M	N
5	P	Q	R	S	T	U	U	W	X	Y	Z	[\]	^	-
6	,	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	}	;	≥	△
8	ç	ü	é	â	ä	à	ç	è	ë	ö	ü	ç	í	ì	ñ	ä
9	É	æ	Æ	ô	ö	ö	û	ÿ	ë	ö	ü	ç	£	¥	®	f
A	á	í	ó	ú	ñ	ñ	é	ç	è	ö	ü	ç	í	ì	»	«
B	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß
C	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł	ł
D	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ
E	æ	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø	ø
F	≡	±	≥	≤	≈	ƒ	÷	≈	°	-	√	≈	≈	≈	≈	≈

8 bits ne suffisent pas...



- Avec ce système, il faut **choisir une page de code** pour l'affichage
Cela empêche par exemple de mélanger du texte grec avec de l'hébreu
- Pour certaines langues – surtout en Asie – il y a sensiblement plus de 256 caractères à coder. **DBCS** (*double byte character set*), où certaines lettres sont codées sur un byte et d'autres sur deux, résout partiellement ce problème
- Avec l'apparition **d'internet**, les ordinateurs doivent communiquer entre eux, et tous ces systèmes se révèlent peu compatibles
Ce qui conduit à l'invention **d'Unicode**



- Ensemble de caractères unique incluant tous les systèmes d'écriture de notre planète, voire d'ailleurs (le Klingon, par exemple:)
 - Le **consortium** Unicode assigne un **nombre unique** à chaque **lettre** de chaque alphabet
 - la lettre anglaise A reçoit le code U+0041, comme en ASCII
 - la lettre arabe Ayn
 - On trouve toutes ces codes sur <http://www.unicode.org>
 - Unicode **évolue**. La version 13.0 sortie en mars 2020 ajoute 5390 caractères
 - Cette version ajout des caractères arabes pour écrire le haoussa, le wolof et d'autres langues en Afrique, et d'autres ajouts pour écrire le hindko et le panjabi au Pakistan. Un caractère a été ajouté pour Syloti Nagri, un script en voie de disparition utilisé pour écrire la langue Sylheti en Asie du Sud, et un autre pour Bopomofo, un alphabet créé pour être utilisé dans la transcription du mandarin à des fins pédagogiques et didactiques pour le cantonais.



- Connaître la représentation Unicode ne suffit pas.
- La chaîne **Hello** correspond à **U+0048 U+0065 U+006C U+006C U+006F**, mais comment stocker ces valeurs en mémoire ?
- L'idée originale était de stocker cela sur 16 bits, mais cela ne suffit pas, il y a maintenant plus de 65535 caractères Unicode.
- 3 encodages coexistent aujourd'hui
 - **UTF-32**, utilisé sous Linux et Unix, code sur 32 bits
 - **UTF-16**, utilisé par Java et Windows, code normalement sur 16 bits, parfois sur 32.
 - **UTF-8**, le plus courant sur internet, utilise entre 8 et 32 bits par caractère. Inclut le code ASCII pour les caractères U+0000 à U+007F



- Pour supporter cela, C++ propose 3 autres types de caractères en plus de `char`
 - `wchar_t` pour les caractères larges (dépend du système)
 - `char16_t` pour l'UTF-16 (C++11)
 - `char32_t` pour l'UTF-32 (C++11)
- A chacun des ces types correspond un type de chaîne à la place de `string`
 - `wstring`
 - `u16string` (C++11)
 - `u32string` (C++11)
- Quant à l'UTF-8, la librairie standard propose de le stocker dans une `string` normale, mais la manipulation des codes de taille variable est plus complexe



- Les préfixes L, u et U permettent de spécifier le type de caractère ou de chaîne
- Le préfixe u8 spécifie que ce qui suit est encodé en UTF-8

```
'A';                                // char
L'A';                                 // wchar_t
u'A';                                 // char16_t
U'A';                                 // char32_t

"hello";                               // const char*
L"hello";                             // const wchar_t*
u"hello";                             // const char16_t*
U"hello";                             // const char32_t*
```



- Le préfixe `R` (pour raw) permet d'écrire des chaînes contenant des `"` et des `\`. La chaîne s'entoure de `"()"` et non `" "`.

```
cout << R("Hello \ " \\ world") << endl;  
// affiche "Hello \ " \\ world"
```

- À partir de C++14, le **suffixe s** indique que la constante est de type `string` et non `const char*`

```
"hello"s;           // std::string  
L"hello"s;         // std::wstring
```



- Pour l'affichage, il faut noter que le flux `cout` ne comprend pas les caractères codés sur plus de 8 bits

```
cout << 'A' << endl;           // A
cout << L'A' << endl;           // 65 en C++17
                                    // ne compile pas en C++20
cout << "hello" << endl;         // hello
cout << L"hello" << endl;        // 0x10000cbc0 par exemple en C++17
                                    // ne compile pas en C++20
```

- Le flux `wcout` résout ce problème en affichant tant les `char` que les `wchar_t`

```
wcout << 'A' << endl;           // A
wcout << L'A' << endl;           // A
wcout << "hello" << endl;         // hello
wcout << L"hello" << endl;        // hello
```

- Il n'existe malheureusement pas de flux équivalent pour `char16_t` et `char32_t`



- La librairie `<locale>` permet d'accéder aux informations de localisation disponibles sur l'OS.
 - Changer la page de code utilisée
 - Demander le symbole monétaire, le symbole de virgule des nombres réels, ...
- Elle redéfinit des versions locales des fonctions de `<cctype>`