



# ***Chaînes de caractères***



- Etre capable de lire, écrire et manipuler du texte
  - sous forme de caractères individuels avec le type `char` et les fonctions de la bibliothèque `<cctype>`
  - sous forme de chaînes de caractères avec le type `string`
  - via les flux d'entrée et de sortie standard, ou les flux vers des chaînes
- Recevoir une brève introduction aux caractères étendus et à la localisation

# HE<sup>VD</sup> IG Plan du chapitre 6



1. Type char [4-14]
2. Chaînes littérales [15-18]
3. string vs vector<char> [19-23]
4. Classe string [24-32]
5. Opérateurs et méthodes [33-59]
6. Flux d'entrée / sortie [60-68]
7. Au-delà du codage ASCII [69-79]
8. Résumé [80-82]



# 1. Type char

# HE<sup>VD</sup> IG Le type char



- Le type `char` permet de stocker des **entiers**, signés ou pas, stockés sur **8 bits**
- Il permet surtout de stocker des **caractères** via un **codage** qui fait correspondre caractères et valeurs numériques
- Le code **ASCII** original utilise 7 bits
  - de 0x00 à 0x1F et 0x7F pour les **codes de contrôle**
  - de 0x20 à 0x7E pour les **caractères imprimables**
- des extensions au code ASCII utilisent
  - 0x80 à 0xFF pour coder les caractères **spéciaux**

# HE<sup>VD</sup> IG Code ASCII



Codes de contrôle

Chiffres

Lettres majuscules

Lettres minuscules

Caractères spéciaux

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ä	231	b
001	Start Of Header	034	"	067	C	100	d	133	å	166	ª	199	Å	232	þ
002	Start Of Text	035	#	068	D	101	e	134	ä	167	º	200	ä	233	ú
003	End Of Text	036	\$	069	E	102	f	135	ç	168	¸	201	å	234	ü
004	End Of Transmission	037	%	070	F	103	g	136	è	169	©	202	ä	235	ù
005	Enquiry	038	&	071	G	104	h	137	é	170	¬	203	å	236	ý
006	Acknowledge	039		072	H	105	i	138	ê	171	½	204	å	237	ÿ
007	Bell	040	(	073	I	106	j	139	ï	172	¾	205	=	238	~
008	Backspace	041	)	074	J	107	k	140	î	173	¿	206	å	239	·
009	Horizontal Tab	042	*	075	K	108	l	141	í	174	«	207	å	240	-
010	Line Feed	043	+	076	L	109	m	142	Ä	175	»	208	ö	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Å	176		209	ö	242	_
012	Form Feed	045	-	078	N	111	o	144	É	177		210	ë	243	¼
013	Carriage Return	046	.	079	O	112	p	145	Ê	178		211	ë	244	½
014	Shift Out	047	/	080	P	113	q	146	Æ	179		212	ë	245	§
015	Shift In	048	0	081	Q	114	r	147	ø	180	¡	213	ì	246	+
016	Delete	049	1	082	R	115	s	148	ó	181	Â	214	í	247	×
017	-- frei --	050	2	083	S	116	t	149	ô	182	Ã	215	î	248	*
018	-- frei --	051	3	084	T	117	u	150	õ	183	Ä	216	ï	249	-
019	-- frei --	052	4	085	U	118	v	151	ù	184	Å	217		250	.
020	-- frei --	053	5	086	V	119	w	152	ý	185	Æ	218		251	'
021	Negative Acknowledge	054	6	087	W	120	x	153	Ö	186		219		252	:
022	Synchronous Idle	055	7	088	X	121	y	154	Ü	187		220		253	;
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188		221		254	
024	Cancel	057	9	090	Z	123	{	156		189		222		255	
025	End Of Medium	058	:	091	[	124		157		190		223			
026	Substitute	059	;	092	\	125	}	158		191		224			
027	Escape	060	<	093	]	126	~	159		192		225			
028	File Separator	061	=	094	^	127		160		193		226			
029	Group Separator	062	>	095	_	128		161		194		227			
030	Record Separator	063	?	096	`	129		162		195		228			
031	Unit Separator	064	@	097	a	130		163		196		229			
032		065	A	098	b	131		164		197		230			

# HE<sup>VD</sup> IG Le type char

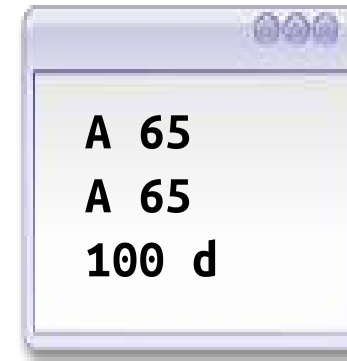


- Le type `char` peut être précédé d'un **modificateur**
  - `signed char` pour des entiers entre **-128 et 127**
  - `unsigned char` pour des entiers entre **0 et 255**
- Sans modificateur, le type peut être **signé ou pas**, cela **dépend du compilateur**, mais les types `char`, `signed char` et `unsigned char` sont trois types distincts
- 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)



- 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;
```



- Attention aux promotions de type !






# Quelques constantes particulières


<code>\n</code>	passage à la ligne
<code>\r</code>	retour chariot « CR »
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\a</code>	alerte sonore
<code>\\</code>	barre oblique inverse « \ »
<code>\'</code>	caractère apostrophe « ' »
<code>\"</code>	caractère guillemet « " »
<code>\b</code>	retour arrière « backspace »
<code>\f</code>	saut de page « form feed »
<code>\13</code>	caractère de code 13 (octal)
<code>\x13</code>	caractère de code 13 (hexadécimal)

# HE<sup>VD</sup> IG Codes de contrôle

```
// afficher la position d'un échappement
cout << "Bell          : " << (int)'\a' << endl;
cout << "Backspace     : " << (int)'\b' << endl;
cout << "Hor Tab        : " << (int)'\t' << endl;
cout << "Line Feed      : " << (int)'\n' << endl;
cout << "Vert Tab       : " << (int)'\v' << endl;
cout << "Carriage R    : " << (int)'\r' << endl;
```



<i>Bell</i>	: 7
<i>Backspace</i>	: 8
<i>Hor Tab</i>	: 9
<i>Line Feed</i>	: 10
<i>Vert Tab</i>	: 11
<i>Carriage R</i>	: 13



000	NUL
001	Start Of Header
002	Start Of Text
003	End Of Text
004	End Of Transmission
005	Enquiry
006	Acknowledge
007	Bell
008	Backspace
009	Horizontal Tab
010	Line Feed
011	Vertical Tab
012	Form Feed
013	Carriage Return
014	Shift Out
015	Shift In
016	Delete
017	-- frei --
018	-- frei --
019	-- frei --
020	-- frei --
021	Negative Acknowledge
022	Synchronous Idle
023	End Of Transmission Block
024	Cancel
025	End Of Medium
026	Substitute
027	Escape
028	File Separator
029	Group Separator
030	Record Separator
031	Unit Separator
032	



# Caractères – catégories

- La bibliothèque `<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)



# Caractères – catégories

```
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



# Caractères – catégories

- <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 **guillemets typographiques** `"Hello, World!"`
  - Pour écrire le caractère `"` au milieu d'une chaîne, on le précède d'une barre oblique inverse, `"\"`
  - 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

```
"partie 1 " "partie 2"
```





# Chaînes de caractères constantes

```
cout << "Ceci est une chaine" << endl;  
  
cout << "" << endl;           // ceci est une chaîne vide  
  
cout << "Ceci n'est qu'une \  
      seule chaine" << endl; // les espaces sont dans la chaîne  
  
cout << "Ceci n'est qu'une "  
      "seule chaine" << endl;
```

Ceci est une chaine

Ceci n'est qu'une                      seule chaine

Ceci n'est qu'une seule chaine



- Notons que ces constantes littérales sont de type `const char*` et non du type `string` introduit en PRG1 au chapitre 2
- Ce type `const char*` sera étudié en PRG2 au chapitre 4 dans le cadre de votre apprentissage du langage C
- En pratique, on peut presque toujours ignorer ce problème, vu qu'il y aura `conversion implicite` vers le type `string` si nécessaire
- Il est toujours possible d'effectuer une `conversion explicite`

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



### 3. `string` vs `vector<char>`

# HE<sup>VD</sup> IG string vs vector<char>



- Pourquoi utiliser `string` et non un `vector<char>` ?
- 99% de ce que vous pouvez faire avec `vector<char>`, vous pouvez le faire avec `string` avec une syntaxe identique
- Le 1% restant inclut...
  - accéder à `.data()` en écriture
  - utiliser `.emplace(...)` et `.emplace_back(...)`
  - utiliser un allocator autre que `std::allocator`
- Mais `string` offre des fonctionnalités spécifiques supplémentaires

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

# HE<sup>VD</sup> IG string ajoute à vector<char>



- Les opérateurs d'affichage << et lecture >>
- Les opérateurs de concaténation + et +=
- Des méthodes synonymes telles que
  - .length() équivalente à .size()
  - .c\_str() équivalente à .data()
- La notion de sous-chaîne, via la méthode .substr(position, longueur)

# HE<sup>VD</sup> IG string ajoute à vector<char>



- Des **surcharges** supplémentaires du constructeur et des méthodes `.assign(...)`, `.erase(...)` et `.insert(...)`
  - Utilisant **position** et **longueur** au lieu des itérateurs
  - Interfaçant avec les **chaînes de type C**
- Des **méthodes** supplémentaires similaires à `.erase(...)` et `.insert(...)`
  - `.append(...)`, équivalente à `.insert(s.end(), ...)`
  - `.replace(pos, ..., ...)`, équivalente à `.erase(pos, ...).insert(pos, ...)`



# string ajoute à vector<char>

- La compatibilité avec les chaînes de caractère de type C  
i.e. `const char*` via
  - la représentation interne (`'\0'` final)
  - des méthodes telles que `.c_str()` et `.copy(...)`
- Des méthodes de **recherche** telles que
  - `.find(...)` et `.rfind(...)`
  - `.find_first_of(...)` et `.find_last_of(...)`
  - `.find_first_not_of(...)` et `.find_last_not_of(...)`



## 4. Classe string



# HE<sup>VD</sup> IG La classe string



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

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

...

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



- On peut initialiser les variables de type `string` des trois manières habituelles

```
string s = "Hello, World!";  
string s("Hello, World!");  
string s{"Hello, World!"};
```

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

```
string s; // contient la chaîne vide ""
```



# Constructeurs similaires à vector

- Le constructeur de copie

```
string s1 = "Bienvenue";  
string s2(s1); // "Bienvenue"
```

- Le constructeur de portion

```
string s1 = "Bienvenue";  
string s2(s1.begin()+4, s1.begin()+8); // "venu"
```

- Le constructeur de remplissage

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

mais contrairement à vector, il n'y a pas de valeur de remplissage par défaut

```
string s(6); // erreur de syntaxe
```



# Constructeurs supplémentaires

- Le constructeur depuis une chaîne de type C

```
string s("Hello, World!"); // "Hello, World!"
```

qui permet l'initialisation depuis une chaîne littérale.  
Le nombre de caractères à copier est déterminé par  
un caractère `'\0'` final dans le tableau C

- Le constructeur depuis un buffer

```
string hello("Hello, World!", 5);  
// contient la chaîne "Hello" de 5 caractères
```

qui permet de sélectionner les N premiers caractères d'une chaîne littérale.



# Constructeurs supplémentaires

- Le constructeur par sous-chaîne spécifie une **partie d'une autre chaîne**, en donnant (ou pas) la **position** du premier caractère et la **longueur** de la sous-chaîne

```
string s(str, pos, len);
```

crée une chaîne **s** de **len** caractères copiés à partir de la 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**



# Constructeur par sous-chaîne

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

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

string hell (hello, 0, 4);      // contient la chaîne "Hell"
string world (hello, 7, 5);     // contient la chaîne "World"
string world2(hello, 7);       // contient la chaîne "World!"
string hello2(hello);          // contient la chaîne "Hello, World!"
```



- L'initialisation par sous-chaîne sans spécifier le paramètre de longueur est un des rares cas où le **type des constantes littérales** a un impact majeur

```
string hello("Hello, World!");           // contient "Hello, World!"  
  
string world(hello, 7);                  // contient "World!"  
  
string hello2("Hello, World!", 7);       // contient "Hello, "  
  
string world2(string("Hello, World!"), 7); // contient "World!"
```



## std::string::string

C++98

C++11



```
default (1)  string();  
copy (2)    string (const string& str);  
substring (3) string (const string& str, size_t pos, size_t len = npos);  
from c-string (4) string (const char* s);  
from buffer (5) string (const char* s, size_t n);  
fill (6)    string (size_t n, char c);  
range (7)   template <class InputIterator>  
             string (InputIterator first, InputIterator last);  
initializer list (8) string (initializer_list<char> il);  
move (9)    string (string&& str) noexcept;
```

### Construct string object

Constructs a `string` object, initializing its value depending on the constructor version used:





## 5. Opérateurs et méthodes



# Affectation – opérateur =

- 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
```



# Affectation – méthode assign()

- 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 "WWWWW"  
str.assign(6, 'W');
```

# HE<sup>VD</sup> IG Plus formellement...



## std::string::assign

<string>

C++98

C++11

C++14

?

```
string (1)  string& assign (const string& str);
substring (2) string& assign (const string& str, size_t subpos, size_t sublen = npos);
c-string (3) string& assign (const char* s);
buffer (4)  string& assign (const char* s, size_t n);
fill (5)    string& assign (size_t n, char c);
range (6)   template <class InputIterator>
             string& assign (InputIterator first, InputIterator last);
initializer list (7) string& assign (initializer_list<char> il);
move (8)    string& assign (string&& str) noexcept;
```

### Assign content to string

Assigns a new value to the string, replacing its current contents.



# Concaténation – l'opérateur +

- L'opérateur + permet de concaténer deux chaînes.

```
string hello("Hello, ");  
string world("World!");  
string hw1 = hello + world;  
// hw1 contient "Hello, World!"
```

- il peut être utilisé avec une constante littérale

```
string hw2 = "Hello, " + world;  
string hw3 = hello + "World!";  
// hw2 et hw3 contiennent "Hello, World!"
```

- Mais pas avec deux

```
string hw4 = "Hello, " + "World!";  
// erreur de compilation
```





# Concaténation – l'opérateur +

- Il peut aussi être utilisé pour **raccrocher un caractère** en début ou en fin de chaîne

```
string hello("Hello, ");  
string hw5 = hello + 'W'; // hw5 contient "Hello, W"  
string hw6 = 'W' + hello; // hw6 contient "WHello, "
```

- Par contre, il n'est pas possible de concaténer une **string** avec un entier

```
string hw7 = hello + 1; // ne compile pas !!
```





# 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 += ',';           // a le même effet que str = str + ',';  
                      // str contient "Hello,"  
  
str += " World!";     // a le même effet que str = str + " World!";  
                      // str contient maintenant "Hello, World!"
```



# Concaténation – la méthode append

- La méthode `.append` offre plus d'options que les opérateurs. Elle s'invoque avec les même 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 "WWWWW"  
str.append(6, 'W');
```



# HE<sup>VD</sup> IG Plus formellement...



## std::string::append

<string>

C++98

C++11

C++14



```
string (1)  string& append (const string& str);  
substring (2)  string& append (const string& str, size_t subpos, size_t sublen = npos);  
c-string (3)  string& append (const char* s);  
buffer (4)  string& append (const char* s, size_t n);  
fill (5)  string& append (size_t n, char c);  
range (6)  template <class InputIterator>  
            string& append (InputIterator first, InputIterator last);  
initializer list (7)  string& append (initializer_list<char> il);
```

### Append to string

Extends the `string` by appending additional characters at the end of its current value:



# Accès aux caractères – opérateur [ ]

Pour une chaîne `str` et un entier `i`, l'expression `str[i]` permet d'accéder en lecture comme en écriture au  $i^{\text{ième}}$  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;
```



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



# Accès aux caractères – méthode at()

- L'opérateur `[]` peut faire crasher le programme 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  $i^{\text{ième}}$  caractère en vérifiant que  $i$  est dans le bon intervalle  
Elle lance une exception spécifique sinon (chap 9)

```
string hello("Hello, World!");  
char fifth  = hello.at(4);  
hello.at(4) = ' '  
  
cout << hello << endl;  
cout << fifth << " remplacé par un blanc" << endl;
```



# Taille d'une string

- On peut demander la **longueur** d'une **string** avec les méthodes **.length()** ou **.size()**
- La valeur retournée 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 hello("Hello");  
size_t longueur = hello.length(); // 5  
size_t taille   = hello.size();    // 5 aussi
```

- La méthode **.empty()** indique si la string est **vide**.

```
bool vide = hello.empty(); // false
```



# Modification de la taille

- On peut modifier la taille d'une chaîne avec la méthode `.resize(len, car)`
  - `len` spécifie la **nouvelle taille**
  - `car` spécifie le **caractère** utilisé pour compléter une chaîne dont la taille augmente. Le caractère nul `'\0'` est utilisé par défaut

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



- La méthode `.substr(pos, 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 str("Hello, World!");
string sub;           // ""
sub = str.substr(0, 5); // "Hello"
sub = str.substr(7, 5); // "World"
sub = str.substr(7);   // "World!"
sub = str.substr();    // "Hello, World!"
```

# HE<sup>VD</sup> IG Insertion



- La méthode `.insert(pos, str)` insère la chaîne `str` en position `pos`
- Augmente la taille de la chaîne sauf si `str` est ""
- `str` peut être remplacé comme pour la méthode `append`

```
string str("to be question"),  
        str2("the "),  
        str3("for not to be");
```

```
str.insert(6, str2);           // to be (the )question  
str.insert(6, str3, 1, 7);     // to be (or not )the question  
str.insert(13, "that is cool", 8); // to be or not (that is )the question  
str.insert(13, "to be ");      // to be or not (to be )that is the question  
str.insert(18, 1, ':');        // to be or not to be(:) that is the question
```

# HE<sup>VD</sup> IG Plus formellement...



## std::string::insert

<string>

C++98	C++11	C++14	?
<i>string</i> (1)	string& insert (size_t pos, const string& str);		
<i>substring</i> (2)	string& insert (size_t pos, const string& str, size_t subpos, size_t sublen = npos);		
<i>c-string</i> (3)	string& insert (size_t pos, const char* s);		
<i>buffer</i> (4)	string& insert (size_t pos, const char* s, size_t n);		
<i>fill</i> (5)	string& insert (size_t pos, size_t n, char c); iterator insert (const_iterator p, size_t n, char c);		
<i>single character</i> (6)	iterator insert (const_iterator p, char c);		
<i>range</i> (7)	template <class InputIterator> iterator insert (iterator p, InputIterator first, InputIterator last);		
<i>initializer list</i> (8)	string& insert (const_iterator p, initializer_list<char> il);		

### Insert into string

Inserts additional characters into the `string` right before the character indicated by *pos* (or *p*):



# HE<sup>VD</sup> IG Remplacement



- `.replace(pos, len, 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 base = "this is a test string.";
string str2 = "n example";
string str3 = "sample phrase";

// Positions:
string s = base;
s.replace(9, 5, str2);
s.replace(19, 6, str3, 7, 6);
s.replace(8, 10, "just a");
s.replace(8, 6, "a shorty", 7);
s.replace(22, 1, 3, '!');

0123456789*123456789*12345
// "this is a test string."
// "this is an example string."
// "this is an example phrase."
// "this is just a phrase."
// "this is a short phrase."
// "this is a short phrase!!!"
```

# HE<sup>VD</sup>IG Plus formellement...



## std::string::replace

<string>

C++98	C++11	C++14	?
<i>string</i> (1)	string& replace (size_t pos, size_t len, const string& str); string& replace (const_iterator i1, const_iterator i2, const string& str);		
<i>substring</i> (2)	string& replace (size_t pos, size_t len, const string& str, size_t subpos, size_t sublen = npos);		
<i>c-string</i> (3)	string& replace (size_t pos, size_t len, const char* s); string& replace (const_iterator i1, const_iterator i2, const char* s);		
<i>buffer</i> (4)	string& replace (size_t pos, size_t len, const char* s, size_t n); string& replace (const_iterator i1, const_iterator i2, const char* s, size_t n);		
<i>fill</i> (5)	string& replace (size_t pos, size_t len, size_t n, char c); string& replace (const_iterator i1, const_iterator i2, size_t n, char c);		
<i>range</i> (6)	template <class InputIterator> string& replace (const_iterator i1, const_iterator i2, InputIterator first, InputIterator last);		
<i>initializer list</i> (7)	string& replace (const_iterator i1, const_iterator i2, initializer_list<char> il);		

### Replace portion of string

Replaces the portion of the string that begins at character *pos* and spans *len* characters (or the part of the string in the range between [*i1*,*i2*)) by new contents:

<http://www.cplusplus.com/reference/string/string/replace>

# HE<sup>VD</sup> IG **Suppression**



- La méthode `.clear()` vide la chaîne
- La méthode `.erase(pos, 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 str("This is an example sentence.");  
str.erase(9, 9); // "This is a sentence."  
str.erase(13);   // "This is a sen"  
str.erase();     // ""
```

# HE<sup>VD</sup> IG Plus formellement...



## std::string::erase

<string>

C++98

C++11



```
sequence (1)  string& erase (size_t pos = 0, size_t len = npos);  
character (2) iterator erase (const_iterator p);  
range (3)   iterator erase (const_iterator first, const_iterator last);
```

### Erase characters from string

Erases part of the `string`, reducing its `length`:



- La plupart de ces méthodes ont comme **valeur de retour** une référence vers **la chaîne elle-même**
- Cela permet **d'enchaîner ces opérations**

```
string maChaine = "123456789";  
cout << "longueur : "  
      << maChaine.append(4, 'a').length();
```

longueur : 13



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

<b>find</b>	Trouve une sous-chaîne
<b>rfind</b>	Idem depuis la fin
<b>find_first_of</b>	Trouve le premier caractère parmi une liste
<b>find_last_of</b>	Idem depuis la fin
<b>find_first_not_of</b>	Trouve le premier caractère hors d'une liste
<b>find_last_not_of</b>	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()`)

# HE<sup>VD</sup> IG find (rfind)



- `.find(str, pos)` trouve la première (dernière) occurrence d'une 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

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

size_t first_needle    = s.find(s2);
size_t second_needle   = s.find(s2, first_needle + 1);
size_t first_haystack  = s.find("haystack");
first_needle           = s.find("needles are small", 0, 6);
second_needle          = s.find("needles are small", first_needle + 1, 6);
size_t first_point     = s.find('.');

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

There are two prepositions in this haystack with needles.



# HE<sup>VD</sup> IG Plus formellement...



## std::string::find

<string>

C++98

C++11



```
string (1) size_t find (const string& str, size_t pos = 0) const noexcept;  
c-string (2) size_t find (const char* s, size_t pos = 0) const;  
buffer (3) size_t find (const char* s, size_t pos, size_type n) const;  
character (4) size_t find (char c, size_t pos = 0) const noexcept;
```

C++11

### Find content in string

Searches the `string` for the first occurrence of the sequence specified by its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences that include characters before *pos*.

Notice that unlike member `find_first_of`, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.





# find\_first\_of (find\_last\_of)

- `.find_first_of(str, pos)` trouve la première (dernière) occurrence d'un caractère de la 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

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

cout << s.find_first_of(vowels) << endl;           // 1
cout << s.find_first_of(vowels, 5) << endl;         // 7
cout << s.find_first_of("Good Bye!") << endl;       // 1
cout << s.find_first_of("Good Bye!", 0, 4) << endl; // 4
cout << s.find_first_of('x') << endl;
// 'x' n'apparaît pas dans "Hello World!", find_first_of
// retourne donc string::npos, soit 18446744073709551615
```



# find\_first\_not\_of (find\_last\_not\_of)

- `.find_first_not_of(str, pos)` trouve la première (dernière) occurrence d'un caractère hors de la 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

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

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

cout << "Le premier caractere non-alphabétique est "
      << str.at(found)                                // '-'
      << " a la position "
      << found << endl;                               // 27
```



- Le type `string` a de nombreuses autres méthodes
  - pour les **comparaisons**: `compare`, `<`, `>`, `<=`, `>=`, `==`, `!=`, ...
  - Identiques à celles de `vector`: `reserve`, `capacity`, `front`, `back`, `push_back`, `pop_back`, `begin`, `end`, ...
  - liées à la représentation des **chaînes en C** (voir PRG2) – `copy`, `c_str`, `data`



## 6. Flux d'entrée / sortie



- L'affichage d'une **string** s'effectue simplement en utilisant l'opérateur <<
- Le manipulateur **setw** vu au chapitre 2 s'applique également, tout comme **right**, **left**, **setfill**, ...

```
string str("Hello!");  
cout << "0123456789*123456789" << endl;  
cout << left << setw(20) << str << endl;  
cout << right << setw(20) << str << endl;  
cout << setfill('.');  
cout << left << setw(20) << str << endl;  
cout << right << setw(20) << str << endl;
```

```
0123456789*123456789  
Hello!  
Hello!  
Hello!.....  
.....Hello!
```



- **<iostream>** définit d'autres flux de sortie que `cout`
  - `cerr` pour afficher des messages d'erreur
  - `clog` pour afficher des relevés
- L'intérêt peut paraître faible vu que ces flux s'affichent aussi à la `console`, tout comme `cout`
- Il apparaît quand on considère la capacité de l'OS à `rediriger les entrées/sorties` d'un programme:

```
prog.exe < in.txt 1> out.txt 2> error.txt
```

- Exécute `prog.exe`
  - lit les entrées depuis le fichier `in.txt`
  - affiche la sortie de `cout` dans `out.txt`
  - affiche les sorties de `cerr` et `clog` dans `error.txt`



- La **lecture** d'une **string** se fait avec l'opérateur **>>**
  - ignore les blancs (espace, tab, retour à la ligne) en début de lecture
  - lit dès le premier caractère non blanc
  - arrête de lire dès le blanc suivant
- Ce comportement standard pour tout type de données n'est **pas toujours approprié** quand on lit une **string**

```
string str;  
cin >> str;           // l'utilisateur entre "James Bond"  
cout << str << endl; // le programme affiche "James"
```

# HE<sup>VD</sup> IG getline()



- La fonction `getline(flux, str)` résout ce problème
  - en lisant toute une ligne depuis `flux`
  - en stockant le résultat dans la chaîne `str`

```
string str;  
getline(cin, str); // l'utilisateur entre "James Bond"  
cout << str << endl; // le programme affiche "James Bond"
```



# HE<sup>VD</sup> IG getline()



- Un troisième paramètre permet de spécifier un caractère de terminaison autre que `'\n'`

```
string str;  
getline(cin, str, '/'); // entrée: "James Bond/John Doe"  
cout << str << endl;   // sortie: "James Bond"
```

- À noter que ce caractère, qu'il soit spécifié ou qu'il soit `'\n'` par défaut, **n'est pas inclus** dans `str`, mais est **supprimé du flux**.  
La lecture suivante commence au caractère qui le suit

# HE<sup>VD</sup> IG `stringstream`



- Le type `stringstream`, défini dans la bibliothèque `<sstream>`, permet de créer un **flux** qui lit ou écrit dans une `string` plutôt que dans la console
- Il peut être initialisé vide ou par constructeur avec une `string`
- L'accès à cette chaîne en lecture ou écriture s'effectue via la méthode `str()`
- Toutes les opérations vues sur `cin` et `cout` sont applicables, ce qui permet par exemple de **convertir** un nombre en chaîne ou inversement



- Conversion de nombre en chaîne

```
int nombre = 123;           // nombre a convertir
stringstream convert;       // flux de conversion
convert << nombre;          // affichage comme pour cout
string chaine = convert.str(); // chaine contient "123"
```

- Conversion de chaîne en nombre

```
string text = "456";        // chaîne a convertir
stringstream convert(text); // flux de conversion
int nombre;
convert >> nombre;          // nombre contient la valeur 456
```



- C++11 a introduit des fonctions simplifiant grandement ces conversions
  - de string à entier avec stoi, stol, stoul, stoll, stoull
  - de string à réel avec stof, stod, stold
  - de nombres à string avec to\_string

```
int    entier = stoi("123");  
double reel   = stod("3.14");  
string chaine = to_string(entier) + " " + to_string(reel);  
// chaine contient "123 3.140000"  
// to_string ne permet pas de choisir le format
```



## 7. Au-delà du codage ASCII



- À 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
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	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	{		}	~	DEL

- ג

[illegible]

# 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
- Le consortium Unicode donne un nombre unique à chaque symbole
  - la lettre anglaise A reçoit le numéro U+0041, codé en UTF8 sur 1 octet de valeur \x41 comme en ASCII
  - la lettre arabe Ayn ع reçoit le numéro U+0639, codé en UTF8 sur 2 octets, de valeur \xd8 + \xb9
  - On trouve tous ces codes sur <https://www.utf8-chartable.de/>
- Unicode évolue. La version 13.0 sortie en mars 2020 ajoute 5390 caractères



- 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 bibliothèque standard propose de le stocker dans une `string` normale, mais la manipulation des codes de taille variable est plus complexe



# Constantes littérales

- 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*
string s; s += '\xd8'; s += '\xb9'; // s contient "ε"
```



# Constantes littérales

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

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

- En 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
cout << "hello" << endl;       // hello
cout << L"hello" << endl;      // 0x10000cb0
cout << s << endl;             // ε avec console utf8
```

- 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 bibliothèque <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>



## 8. Résumé





- Le type `char` permet de stocker un caractère unique, typiquement codé sur 8 bits en `ASCII`
- `<cctype>` fournit des fonctions permettant de manipuler ces caractères
- Le type `string` permet de stocker et manipuler des chaînes de caractères
- L'initialisation par constructeur offre de nombreuses options supplémentaires
- Les opérateurs et les méthodes permettent de concaténer, redimensionner, extraire/insérer/remplacer/supprimer des sous-chaînes, ou d'effectuer des recherches



- Les **flux d'entrée/sortie** permettent de lire et écrire les chaînes  
La lecture d'une chaîne contenant des blancs s'effectue avec `getline`
- `stringstream` permet de définir un flux lisant/écrivain dans une string  
C'est une des méthodes possibles pour **convertir** des nombres en chaînes de caractères et vice-versa
- 8 bits ne suffisent pas pour représenter tous les caractères possibles  
**Unicode** regroupe tous ces caractères, mais **l'encodage** des valeurs Unicode peut varier (**UTF-8**, 16, 32).
- C++ offre des types adaptés de `char` et de `string`