

Onboarding - Recap 03 - Librería Estándar de C++

Collapse context

◀▶

C++ tiene una librería estándar muy completa. No es tan extensa como la de Python pero aun así **es un buen arsenal a tu disposición**.

Este *recap* es para darte un pantallazo de la librería.

El objetivo **no** es que memorizes sino que **veas** de que es capaz C++ y sepas en donde podrás encontrar **buena** información sobre la librería.

Cuanto más sepas, más **herramientas** tendrás para resolver el TP (*y cuanto menos codees, mejor!*).

Nota: tal vez quieras hacer primero "Recap - Proceso de Building y Testing" y "Recap - Memoria en C++" antes de seguir.

Your answer passed the tests! Your score is 100.0%. [Submission #64e9070f000849d95010b5c2]

✕

Question 1:

`std::string` tiene varios métodos para trabajar con texto.

```
std::string foo = "hola mundo";
std::string bar = foo.substr(2, 4);
```

Que valor tiene el string `bar`?

Referencias:

- [substr](#)

☐ "hola"

☒ "la m"

☐ "la"

☐ "ola "

Question 2:

Tanto `std::string` como `std::vector<char>` pueden trabajar con datos binarios sin embargo varios métodos de `std::string` dependen de si se esta trabajando con texto o con binario, algunos son propenso a errores y otros no pueden usarse directamente (y es fácil caer en ellos por error)

Por eso es recomendable usar **siempre** `std::vector<char>` cuando se trabaja con **datos binarios** ya que sus metodos no asumen (o imponen) restriccion alguna sobre el contenido (lease, no suponene que hay un `\0` al final)

Cuando se trabaja con **texto** `std::string` esta perfecto y es preferible por sobre `std::vector<char>`

```
// Suponer que 'p' es un char* y que 'sz' es un size_t

std::string s1(p);
std::string s2(p, sz);

std::string s3;
s3.append(p);

std::string s4;
s4.append(p, sz);

std::string s5(p, strlen(p));

std::string s6(p, sz);
std::string s7(s6.c_str());
```

Cuales de estas afirmaciones son correctas?

Cuidado que hay múltiples respuestas correctas! **Marcarlas todas!**

Referencias:

- [string](#)
 - [append](#)
 - [c_str](#)
- ☒ **c_str()** retorna un string que termina en `\0` por lo tanto `s7` se copia de él. Si y solo si `p` es texto entonces `s7` es una copia exacta de `s6`.
- ☒ Si `p` es un texto (no binario) que termina en un `\0`, `s1` se inicializa copiando a `p` pero si `p` es binario (puede contener multiples `\0` en su interior), `s1` esta indefinido.
- ☒ Si `p` es binario (puede o no contener múltiples `\0`) entonces al final de `s4` se copian `sz` bytes de `p`
- ☒ Si `p` es binario entonces `s2` se inicializa copiando `sz` bytes de `p`.
- ☒ Si `p` es binario (puede o no contener múltiples `\0`) entonces `s3` luego de llamar `append` queda indefinido.
- ☒ Si `p` es texto (finaliza con un `\0`) entonces al final de `s3` se copia de `p` hasta encontrar el `\0`
- ☐ Si `p` es texto (finaliza con un `\0`) entonces `s2` se inicializa copiando `sz` bytes de `p` o hasta encontrar el `\0`
- ☐ Si `p` es texto (finaliza con un `\0`) entonces `s5` se inicializa copiando `p` incluyendo el `\0` al final.
- ☐ **c_str()** retorna un string que termina en `\0` por lo tanto `s7` se copia de él. Independientemente de si `p` es texto o binario, `s7` es una copia exacta de `s6`.
- ☒ Si `p` es texto (finaliza con un `\0`) entonces `s1` se inicializa copiando `p` incluyendo el `\0` al final.

Question 3:

```
// Suponer que 'p' es un char* y que 'sz' es un size_t

std::vector<char> v1(p);
std::vector<char> v2(p, p+sz);

std::vector<char> v3;
v3.insert(v3.end(), p, p+sz);


std::vector<char> v4(v2.data(), v2.size());
```


Cuales de estas afirmaciones son correctas?

Cuidado que hay múltiples respuestas correctas! **Marcarlas todas!**


Information	
Author(s)	Martin Di Paola
Deadline	05/09/2023 18:00:00
Status	Succeeded
Grade	100%
Grading weight	1.0
Attempts	23
Submission limit	No limitation


Submitting as

 AbrahamOsco 102256

 [Classroom : Default classroom](#)

For evaluation

 Best submission

 [25/08/2023 16:54:55 - 100.0%](#)

Submission history

25/08/2023 16:54:55 - 100.0%
24/08/2023 23:30:05 - 93.75%

Referencias:

- [vector](#)
 - [insert](#)
 - [data](#)
- ☒ Independientemente de si `p` es texto o binario, se copian `sz` bytes de `p` y se los inserta al final del vector `v3`.
- ☒ `v2` se inicializa con el contenido de `p` copiando `sz` bytes de `p` este o no terminado en un `\0`.
- ☒ `data()` retorna un puntero al contenido del vector `v2` por lo tanto `v4` se copia de él tantos bytes como le son indicados. Siendo estos `v2.size()`, `v4` resulta en una copia exacta de `v2` independientemente de si el contenido tiene o no `\0`.
- ☐ `v1` se inicializa con el contenido de `p` copiando hasta encontrar un `\0`

Question 4:

`std::string` permite concatenar strings con `+` pero si se quiere armar un string más complejo es más cómodo trabajar con `std::stringstream` ya que permite hacer distintas conversiones hacia string en diferentes formatos.

```
// ok, pero rapidamente tiene sus limitaciones
std::string who = "Alice";
who += " and ";
who += "Bob";

// mucho mas flexible ya q permite formatear a string
// numeros en distintas bases (entre otras cosas)
std::stringstream ss;
ss << who << ": " << 42 << " (" << std::setbase(16) <<
42 << ")";

std::string msg = ss.str()
```

Que valor tiene el string `msg`?

Referencias:

- [substr](#)
 - [stringstream](#)
 - [stringstream::str](#)
 - [setbase](#)
- ☐ "Alice and Bob: 42 (0x10 42)"
- ☐ "Bob: 42 (2a)"
- ☐ "Bob: 42 (16 42)"
- ☐ "Alice and Bob: 42 (16 42)"
- ☐ "Alice and Bob: 42 (0x2a)"
- ☒ "Alice and Bob: 42 (2a)"

Question 5:

✓

Te dije que era tricky. El `push_back` de `std::vector` es más lento que el de `std::list` por que hay reallocs para hacer crecer al vector. Si **pre-allocas** el vector, entonces `std::vector` va a ser más rápido que `std::list`.

✕

C++ nos da gratis estructuras de datos muy eficientes, llamadas contenedores o *containers*.

Ahora, que un programa en C++ sea eficiente **también depende de vos**.

Supone el siguiente código:

```
for (int i = 0; i < 1000; ++i) {
    container.push_back(i);
}
```

Que container tendría una mejor performance? Es una pregunta tricky por que depende de como este inicializado el container así que imaginate que está inicializado totalmente vacío.

- Referencias:
- [std::vector::push_back](#)
 - [std::list::push_back](#)

☒ `std::list<int>`

☐ `std::vector<int>`

Question 6:

Cuales de las siguientes formas de inicializar un `std::vector` deberías usar para **pre-allocar** un vector (con valores dummy)?

```
std::vector<int> first;
std::vector<int> second (1000,33);
std::vector<int> third (second.begin(),second.end());
std::vector<int> fourth (third);
```

Lo confieso, el ejemplo me lo robé de [cplusplus\(ingles\)](#)

Y si, esto es un spoiler de la respuesta de la pregunta anterior.

☐ `fourth`

☐ `first`

☐ `third`

☒ `second`

Question 7:

Que hace este código? (Perdonadme que use nombres de variables **tan poco descriptivas**, pero es que sino sería muy fácil)

```
std::map<char, int> f;
for (auto &c : "It's a sad truth that those who shine
brightest often burn fastest") {
    f[c] += 1;
}

for (auto &kv : f) {
    std::cout << kv.first << ": " << kv.second <<
"\n";
}
```

- Referencias:
- [Que es un std::map](#)
 - [Como funciona el operador \[\] de un std::map](#) (mirate el ejemplo)

- [Como iterar sobre un std::map](#)

- ☒ Imprime la cantidad de veces que aparece cada letra en el string.
- ☐ Imprime las posiciones de cada letra en el string
- ☐ Imprime las letras del string

Question 8:

El siguiente código busca el número **16** en un contenedor usando el algoritmo `std::find`.

C++ no sólo te da estructuras de datos como `std::map` y `std::list` sino que también te da *algoritmos*.

La pregunta es, que container debe ser `foo` para que se pueda usar `std::find`?

```
auto it = std::find(foo.begin(), foo.end(), 16);
if (it != foo.end()) {
    std::cout << "Encontrado!\n";
}
```

- ☐ `foo` puede ser `std::list` o `std::vector` por que permiten recorrer el container desde el principio `begin()` hasta el final `end()`.
- ☐ `foo` puede ser cualquier container menos `std::unordered_map` por que es un container que no tiene un orden definido.
- ☒ `foo` puede ser cualquier container; lo único que importa es que el container tenga al menos iteradores que puedan moverse hacia adelante (*forward*).
- ☐ `foo` debe ser un `std::list` por que es quien soporta tener iteradores.

Question 9:

Cuales de los siguientes algoritmos de C++ te permite saber si un elemento está o no en un container por **búsqueda binaria**?

- Referencias:
- [Listado de los algoritmos](#)

Esta pregunta es trivial: lo único que tenes que hacer es ir a la referencia, apretas *"Ctrl-F"* y buscas *"binary"* :D

Que tal si le das una ojeada por arriba a que hacen otros algoritmos también? Por ejemplo [sort](#), [lower_bound](#), [max](#) y [remove_if](#)

Pensá que **implementar** un algoritmo lleva mucho más tiempo que **usar** uno ya hecho.

Y te garantizo que tiempo es lo que nunca sobra.

- ☐ `std::search`
- ☐ `std::quick_search`
- ☐ `std::find`
- ☒ `std::binary_search`

Question 10:

Que hace este código? Apuesto que vas a usar este truquito para debuggear (lo que sí, no los lla**m**es **z** que son nombres horribles, pone mejores nombres: **usa nombres descriptivos siempre**).

```
#include <iostream>
#include <algorithm>
#include <list>

template<class T>
void m(const T &z) {
    std::cout << z << " ";
}

int main() {
    std::list<int> l = {1984, 2022, 101};
    std::for_each(l.begin(), l.end(), m<int>);
    return 0;
}
```

- ☒ Imprime el contenido del container
- ☐ Imprime... (ok, lo admito, este choice es demasiado obvio y no se me ocurre otras opciones pero no quería dejar pasar el hecho que `std::for_each` te puede simplificar la vida)

Question 11:

El siguiente código comete un **error letal**. Podes compilarlo y ejecutarlo y ver que pasa. Puede que te crashee o puede que no pero seguro que está mal.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lista = {1984, 2022, 101, 33};
    for (auto it = lista.cbegin(); it != lista.cend(); ++it)
        if (*it % 2 == 0) // remover si es par
            lista.erase(it);

    for (int i : lista)
        std::cout << i << "\n";
    return 0;
}
```

Cual es este **error letal** ?

Referencias:

- [std::list::erase](#)
- ☒ Modificar un container mientras se lo itera deja a los iteradores corruptos. Luego de una modificación hay que crear otro iterador.
- ☐ El método `erase` retorna una lista nueva con el elemento borrado. Al no guardar la referencia los cambios se pierden. Hay que hacer `lista = lista.erase(*it);`
- ☐ Se esta usando `cbegin` y `cend` para obtener iteradores que no permitan la modificación del container. Por eso el comportamiento es indefinido. Hay que usar `begin` y `end`.

Question 12:

El siguiente código itera un diccionario en Python y borra las entradas (keys) que sean multiplo de 2.

```
for key in a_dict:
    if key % 2 == 0:
        del a_dict[key]
```

Python? C++ es un lenguaje difícil, no hay duda, pero muchas de las complejidades que veras en Taller no son propias de C++ y pueden pasar incluso en lenguajes de alto nivel como en Python.

Es entonces el código mostrado válido en Python? Se puede modificar un container mientras se lo itera?

Referencias:

- [dict](#)
- ☐ Python es un lenguaje de alto nivel y permite modificar un container mientras se lo itera.
- ☒ Modificar un container mientras se lo itera deja a los iteradores corruptos. El codigo Python no es correcto.

Question 13:

Digamos que tenes un archivo de texto que tiene datos de una persona por línea como el que sigue:

```
alice 33 2001 azul
bob 21 1000 rojo
charlie 25 1200 rojo
```

C++ puede hacerte la vida muy fácil para parsear estos archivos si sabes qué método de `std::fstream` tenes que usar.

Cual de todos estos métodos te permite leer cada campo y cargarlos a variables directamente y de la manera **más simple**?

Referencias:

- [getline](#)
- [operador >>](#)
- [read](#)
- ☐ El método `getline` justamente te permite cargar una línea de texto en un `std::string`. Luego se puede usar `std::string::find` para buscar los espacios que delimitan los campos y parsearlos de a uno.
- ☒ El operador `>>`. Haces algo como `archivo >> nombre >> edad >> numero >> color;` y el operador leera del archivo e ira cargando las variables una a una separadas por espacios.
- ☐ Se hace un `read` de un tamaño lo suficientemente grande para guardar todo el archivo en un buffer `char*` y al estar en memoria se hace un parsing normal.

Question 14:

Se quiere leer de entrada estandar un comando compuesto únicamente por 1 palabra y luego una frase que puede estar compuesta por 1 o más palabras finalizada por un salto de línea que servirá como argumento para el comando.

Por ejemplo `"SEND un texto aqui"`, donde `"SEND"` es el comando y `"un texto aqui"` seria la frase a modo de argumento.

Entre el comando y la frase puede haber uno o más espacios.

Por ejemplo "RECV este es el argumento" daría "RECV" como el comando y "este es el argumento" como la frase.

Cuál de los siguientes códigos parsea correctamente a la entrada estandar y es el más simple?

```
std::string comando, argumento;
std::cin >> comando >> std::ws;
std::getline(std::cin, argumento);
```

```
std::string comando, argumento;
std::cin >> comando >> std::ws >> argumento;
```

```
std::string comando, argumento;
std::cin >> comando;
std::getline(std::cin, argumento);
```

```
std::string comando, argumento;
std::cin >> comando;
char c = std::cin.peek();
while (c == ' ') {
    std::cin.get();
    c = std::cin.peek();
}
std::getline(std::cin, argumento);
```

Ojo! De esos códigos hay varios que funcionan pero uno es claramente más simple.

Recordá que cuanto más simple sea tu código, menos propenso es a tener bugs y menos tiempo tendras que invertir en debuggear.

Y si tenes dudas, copiate el código, compilalo y correlo! Que forma más simple de ver lo q hace un código que viendolo!

Referencias:

- [getline](#)
- [operador >>](#)
- [ws](#)
- [peek](#)

- ☐ Código 4
- ☐ Código 3
- ☒ Código 1
- ☐ Código 2

Question 15:

Se tiene un archivo **binario** (digamos una imagen) y se quiere cargar en memoria los primeros N bytes.

```
char* buf = new char[N];
f.read(buf, N);
```

```
std::vector<char> buf(512, 0);
f.read(&buf[0], 512);
```

```
char* buf = new char[N];
f.getline(buf, 512);
```



```
char* buf = new char[N];  
f.getline(&buf[0], 512);
```

```
std::string buf(512, 0);  
f >> buf;
```

Cuales de estas afirmaciones son correctas?

Cuidado que hay múltiples respuestas correctas! **Marcarlas todas!**

Referencias:

- [getline](#)
- [read](#)
- [Bonus track opcional](#)

- ☒ `getline` es solo para texto, no deberia ser usado en archivos binarios.
- ☒ Reservar memoria con `new` esta bien pero es preferible no hacerlo y usar una estructura de datos como `std::vector` ya que te libera la memoria correctamente.
- ☐ Aunque se puede usar `new`, es preferible usar `std::string` ya que libera los recursos por nosotros y funciona sin problemas con datos binarios.
- ☐ Hacer `&buf[0]` cuando `buf` es de tipo `std::vector<char>` no tiene sentido (no compila).

Question 16:

Armame un mini apunte de las estructuras y algoritmos y cosas que has visto en este recap.

Cuando programes los TPs lo tendras a tu disposición para no tener q reinventar la rueda.

Ayudate leyendo más la documentacion oficial que Stackoverflow.

- ☐ No, no tengo un apunte y no creo necesitarlo. Si necesito de una estructura o algoritmo, prefiero no usar la STL y codear en cambio mi propia versión que será más eficiente y libre de errores.
- ☒ Si, tengo un mini apunte con lo visto en el recap

Submit