Ayudantía 6
Tarea 2

Javier Larré

Pontificia Universidad Católica de Chile
javier.larre@uc.cl

8 de mayo de 2026


Buenos días!!!
Llegó el nuevo día.

Ayudantía de hoy

1. Sobre la T2
2. Intro a C++
3. C++ Avanzado
   - Templates
   - Closures
4. Intro a Ruby
5. Concurrencia
6. Referencing

DCCFarm

1. **Importante:** La evaluación será hecha en base a *test cases*, cada parte de la tarea vale 50%.
2. Manejarás **Funciones de Primer Grado**. Es decir que construirás, entregarás y recibirás funciones dentro de tus
3. Implementarás la interpretación de un lenguaje OOP.
4. **Fecha de entrega:** Martes 26 de Mayo.
5. Esta tarea cuenta con política de atraso hasta 2 días.



Intro a C++

C++ es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C y añadir mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, C++ es un lenguaje híbrido.

Para este curso utilizaremos el compilador *g++* con std23 y compilaremos con el comando `make test`.

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello world!";
    return 0;
}
```

IMPORTANTE

IMPORTANTE!
Se evaluarán **7 de los 9** ejercicios. Puedes ignorar 2 ejercicios que no desees realizar, y si entregas más de 7 ejercicios entonces descontaremos los que tengan mayor puntaje.

Intro a C++

- Sintaxis y type binding similar a Java (y C, obviamente).
- *single quotes* para caracteres, *double quotes* para strings.
- `// cometario` o `/*comentario*/` para comentar código
- Pueden printear con `std::cout << "Hello World!" << std::endl;`
- La memoria es manejada **MANUALMENTE**.

```
int a;
// default-initialization
// Traditional initialization forms
int b = 5;
// copy-initialization
int c ( 6 );
// direct-initialization
// Modern initialization forms
int d { 7 };
// direct-list-initialization
int e {};
// value-initialization
```

Intro a C++

alignas	const	extern	or_eq	throw
alignof	consteval (since C++20)	false	private	true
and	constexpr	float	protected	try
and_eq	constinit (since C++20)	for	public	typedef
asm	const_cast	friend	register	typeid
auto	continue	goto	reinterpret_cast	typename
bitand	co_await (since C++20)	if	requires (since C++20)	union
bitor	co_return (since C++20)	inline	return	unsigned
bool	co_yield (since C++20)	int	short	using
break	decltype	long	signed	virtual
case	default	mutable	sizeof	void
catch	delete	namespace	static	volatile
char	do	new	static_assert	wchar_t
char8_t (since C++20)	double	noexcept	static_cast	while
char16_t	dynamic_cast	not	struct	xor
char32_t	else	not_eq	switch	xor_eq
class	enum	nullptr	template	
compl	explicit	operator	this	
concept (since C++20)	export	or	thread_local	

Recursos

General

https://www.learncpp.com/
https://godbolt.org/
https://en.cppreference.com/

Templates

Templates

- Permiten escribir código **generalizando los tipos**.
- El compilador genera versiones específicas según los tipos usados.
- Se resuelven (en el 99% de los casos) en **tiempo de compilación**.
- Es un `void*` más seguro.

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

template <typename T>
T myMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << myMax(3, 7) << '\n';
    cout << myMax(2.5, 1.8) << '\n';
    cout << myMax('a', 'z') << '\n';
    return 0;
}
```

Class templates

Permiten parametrizar clases por tipo. Util en estructuras de datos, sobre todo simples o de un uso.

`vector<T>, stack<T>, pair<T1, T2>.`

```
template <typename T>
class Stack {
private:
    static const int MAX = 100;
    T data[MAX];
    int topIndex;
        /*...*/
}
```

Specialization

- A veces un caso particular necesita una implementación distinta.
- Eso se logra con **specialization**.
- Útil cuando un tipo requiere tratamiento especial.

```
Horse myMax(Horse a, Horse b) {
    return (a.age > b.age) ? a : b;
}
```

Closures

Closures

- El compilador le asigna un **tipo único**.
- Usen la keyword `auto` cuando las guarden en variables temporales, ya que el tipo definido se puede ver feo.
- Pueden capturar variables del enviroment.
- No deben liberarlas manualmente de la memoria, a menos que las usen en el heap.

```
auto makeAdder(int n) {
    return [n](int x) {
        return x + n;
    };
}
```

std::function

- Sirve para guardar closures con la misma firma.
- Puede almacenar lambdas y funciones normales.
- La sintax es `std::function<ret(args)>`

```
function<int(int)> makeAdder(int n) {
    return [n](int x) {
        return x + n;
    };
}
```

Recursos

Templates

`https://en.cppreference.com/cpp/language/templates`

`std::function`

`https://en.cppreference.com/w/cpp/header/functional`

Lambdas

`https://en.cppreference.com/w/cpp/language/lambda`

Intro a Ruby

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro ”Matz” Matsumoto en 1995. Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU.
Puedes instalarlo en ruby-lang.org.

```ruby
def hello
  "hello"
end

def こんにちは
  puts "means hello in Japanese"
end
```



Intro a Ruby

- Interpretado por la máquina virtual de ruby.
- Dynamic type binding (no necesita declarar tipos) y utiliza un *garbage collector*.
- Las variables de instancia son privadas, se deben acceder con métodos.
- La iteración se puede hacer con `.each`
- Soporta *blocks*, o lambdas, para iteraciones y otras operaciones.
- Tiene algunas diferencias con Python. Por ejemplo los strings son mutables y tiene accessors de atributos (public y private).

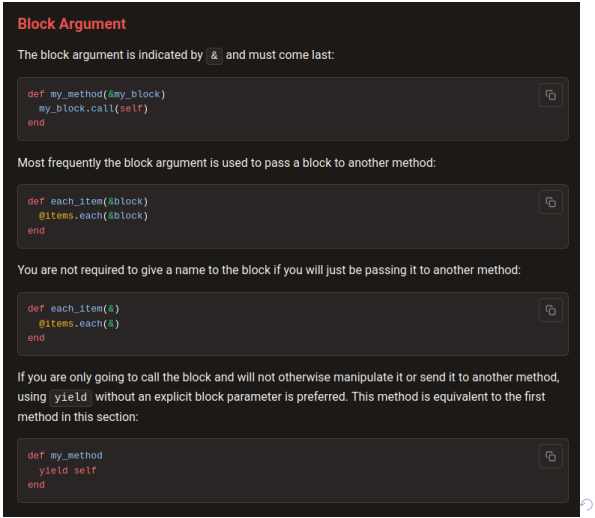
```
class Greeter
  def names
    @names
  end

  def names=(names)
    @names = names
  end

  def say_hi
    if @names.nil?
      puts "..."
    elsif @names.respond_to?("each")
      @names.each do |name|
        puts "Hello␣#{name}!"
      end
    else
```

Yield

- Tiene un significado totalmente diferente que en Python.
- Es para ejecutar un bloque de código (o cualquier *callable*)
- Si necesitan recibir un lambda para manipularlo, almacenarlo o usarlo para otro método entonces deben usar &



Block Argument

The block argument is indicated by `&` and must come last:

```
def my_method(&my_block)
  my_block.call(self)
end
```

Most frequently the block argument is used to pass a block to another method:

```
def each_item(&block)
  @items.each(&block)
end
```

You are not required to give a name to the block if you will just be passing it to another method:

```
def each_item(&)
  @items.each(&)
end
```

If you are only going to call the block and will not otherwise manipulate it or send it to another method, using `yield` without an explicit block parameter is preferred. This method is equivalent to the first method in this section:

```
def my_method
  yield self
end
```

Intro a Ruby

```ruby
interpreter = Interpreter.new

animal_class = ClassDef.new("Animal", nil, [
  MethodDef.new("initialize", ["name"], [
    Assignment.new("@name", Variable.new("name"))
  ]),
  MethodDef.new("speak", [], [
    Literal.new("El animal habla")
  ]),
  MethodDef.new("get_name", [], [
    Variable.new("@name")
  ])
])

interpreter.evaluate(animal_class)

dog_instance = interpreter.evaluate(New.new("Animal", [Literal.new("Perro")]))

speak_result = interpreter.evaluate(MethodCall.new(dog_instance, "speak", []))
# >>> "El animal habla"

name_result = interpreter.evaluate(MethodCall.new(dog_instance, "get_name", []))
# >>> "Perro"

dog_class = ClassDef.new("Dog", "Animal", [
  MethodDef.new("speak", [], [
    Literal.new("Guau!")
  ])
])

interpreter.evaluate(dog_class)

rex = interpreter.evaluate(New.new("Dog", [Literal.new("Rex")]))
speak_rex = interpreter.evaluate(MethodCall.new(rex, "speak", []))
# >>> "Guau!"
```

Recursos

QuickStart

`https://www.ruby-lang.org/en/documentation/quickstart/`

Unit Testing

`https://ruby-doc.org/stdlib-3.1.0/libdoc/test-unit/rdoc/Test/Unit.html`

Documentación

`https://ruby-doc.org/`

Concurrencia



Ejercicios de concurrencia

- Describe el problema productor-consumidor.
- Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
- Cuál es el problema de *busy waiting*?
- Qué es lo que indica la variable `counter` en un semáforo?
- Escribe en pseudocódigo las 2 operaciones de un semáforo (wait y release)
- Qué desventaja trae usar semáforos?
- Qué ocurriría si las operaciones de un semáforo fueran **no atómicas**?

References


Robert Sebesta (2023)
Concepts of programming languages

The End

Questions? Comments?