Estamos corrigiendo la T1!

Publicación de las notas: TBD
Temas de la T2: TBD
Fechas de la T2: TBD
Lenguajes de la T2: TDB



Javier Larré

Control structs

pero primero un poco de punteros

10 de Abril de 2026

IIC2560

PUNTEROS!!!



- ¿Cuál es la diferencia entre un puntero y una referencia?

PUNTEROS!!



- ¿Cuál es la diferencia entre un puntero y una referencia?
  - Una referencia no es una dirección de memoria concreta, solamente se refiere a un objeto o valor en memoria
- ¿Para qué uso es necesario usar punteros?

PUNTEROS!!!



- ¿Cuál es la diferencia entre un puntero y una referencia?
    - Una referencia no es una dirección de memoria concreta, solamente se refiere a un objeto o valor en memoria
- ¿Para qué uso es necesario usar punteros?
    - Principalmente para manejar estructuras dinámicas en memoria. En assembly tambien se usa direccionamiento indirecto.
- Si un lenguaje NO tiene un tipo para punteros o referencias ¿Cómo implementarían un árbol binario?

PUNTEROS!!!

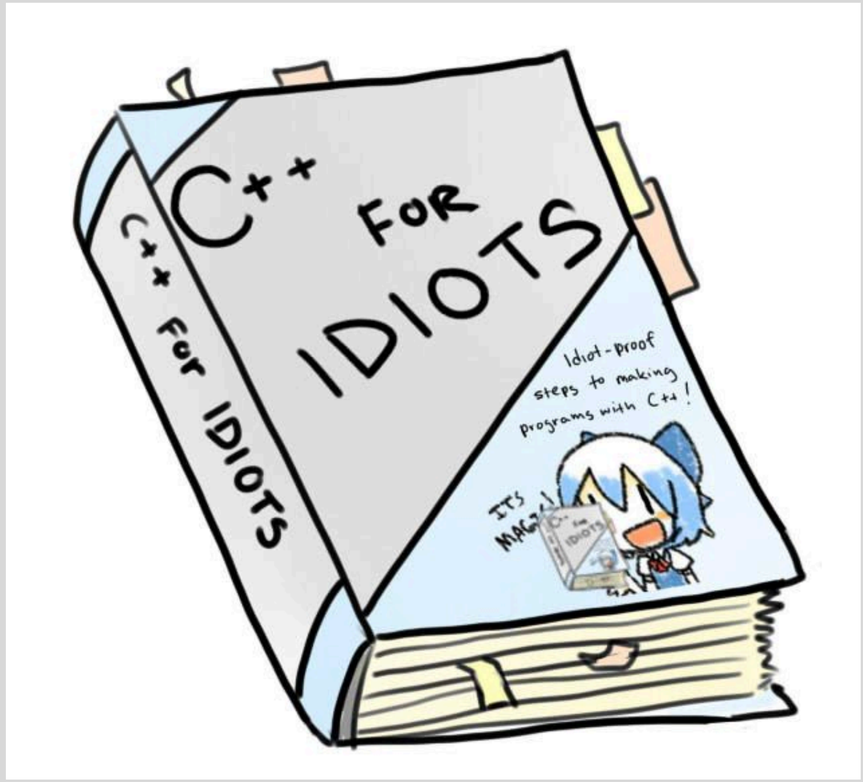


- ¿Cuál es la diferencia entre un puntero y una referencia?
  - Una referencia no es una dirección de memoria concreta, solamente se refiere a un objeto o valor en memoria
- ¿Para qué uso es necesario usar punteros?
  - Principalmente para manejar estructuras dinámicas en memoria. En assembly tambien se usa direccionamiento indirecto.
- Si un lenguaje NO tiene un tipo para punteros o referencias, cómo implementarían un árbol binario?
  - Como mínimo, es necesario tener una *pool* finita de nodos para usar.

El operador ->

¿Cual es la **justificación** de este operador en C y C++?

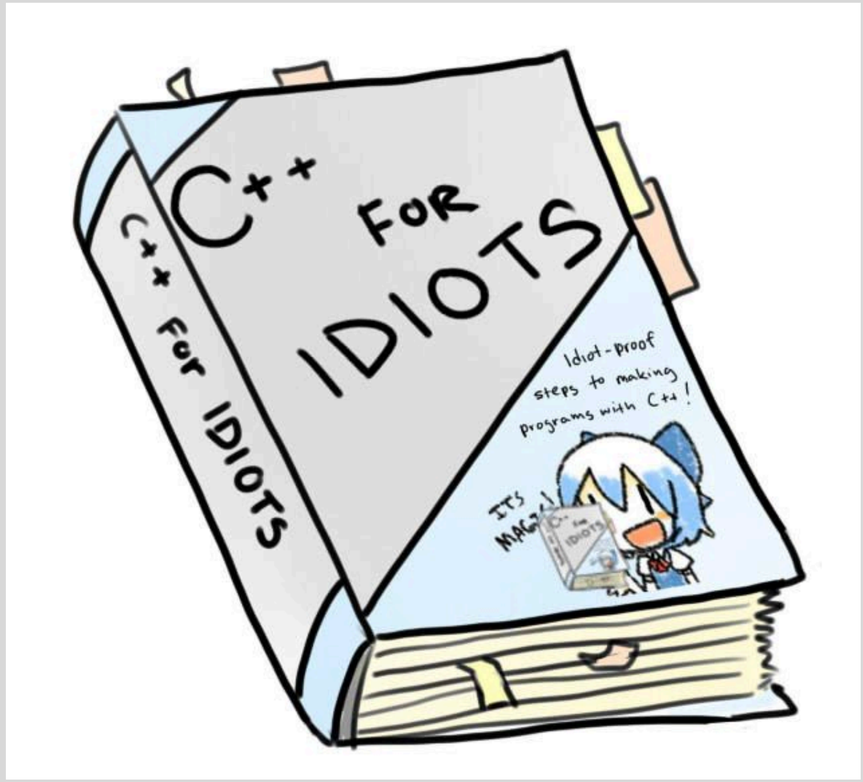
```
struct foo {
  int x;
  float y;
 };
struct foo var;
struct foo* pvar;
pvar = malloc(sizeof(struct foo));
var.x = 5;
(&var)->y = 14.3;
pvar->y = 22.4;
(*pvar).x = 6;
```



El operador ->

¿Cual es la **justificación** de este operador en C y C++?

El operador flecha (->), usado entre un puntero a una estructura y el nombre de una variable o campo de la estructura, dereferencia el puntero y referencia la variable.
Es decir, hace dos pasos con un solo símbolo.
Esto es un ejemplo de *syntatic sugar*



En el lenguaje ALGOL (Burroughs Extended) las matrices son almacenadas como arrays de una dimensión. ¿Cuál es la ventaja de esta implementación? ¿Cuál es la desventaja?

Última pregunta sobre punteros

6.5.6 Rectangular and Jagged Arrays

A **rectangular array** is a multidimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly.

A **jagged array** is one in which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each layer can have a different number of elements. Jagged arrays are made possible when multidimensioned arrays are actually arrays of arrays. For example, a matrix would appear as an array of single-dimensioned arrays.

C, C++, and Java support jagged arrays but not rectangular arrays. In those languages, a reference to an element of a multidimensioned array uses a separate pair of brackets for each dimension. For example,

```
myArray[3][7]
```

C# and F# support both rectangular and jagged arrays. For rectangular arrays, all subscript expressions in references to elements are placed in a single pair of brackets. For example,

```
myArray[3, 7]
```


To get an actual address value, the number of elements that precede the desired element must be multiplied by the element size. Now, the access function can be written as

location(a[i,j]) = address of a[0, 0]
+ ((((number of rows above the *ith* row) * (size of a row))
+ (number of elements left of the *jth* column)) *
element size)

Because the number of rows above the *ith* row is i and the number of elements to the left of the *jth* column is j, we have

location(a[i, j]) = address of a[0, 0] + (((i * n) + j) *
element_size)

where n is the number of elements per row. The first term is the constant part and the last is the variable part.

The generalization to arbitrary lower bounds results in the following access function:

location(a[i, j]) = address of a[row_lb, col_lb]
= (((i − row_lb) * n) + (j − col_lb)) * element_size

where row_lb is the lower bound of the rows and col_lb is the lower bound of the columns. This can be rearranged to the form

location(a[i, j]) = address of a[row_lb, col_lb]
− (((row_lb * n) + col_lb) * element_size)
+ (((i * n) + j) * element_size)

where the first two terms are the constant part and the last is the variable part. This can be generalized rather easily to an arbitrary number of dimensions.

For each dimension of an array, one add and one multiply instruction are required for the access function. Therefore, accesses to elements of arrays with several subscripts are costly. The compile-time descriptor for a multidimensional array is shown in Figure 6.6.

Javier Larré

Control structs

Ahora si

10 de Abril de 2026

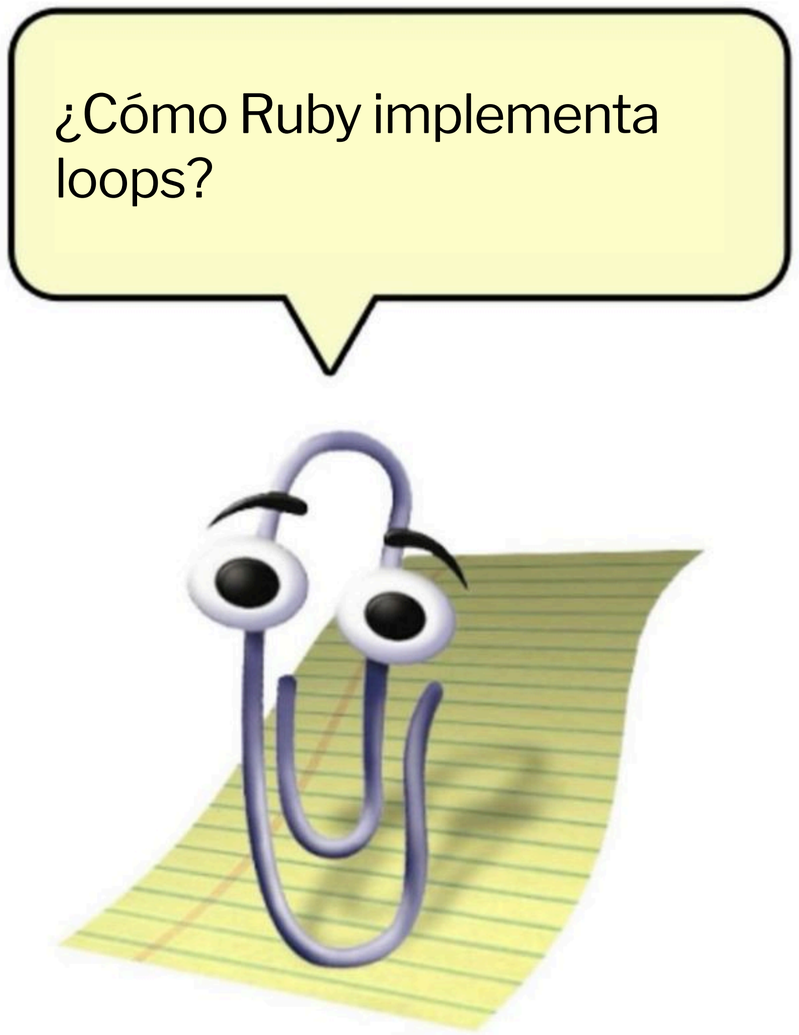
IIC2560

En el caso de los switch case, es posible que se ejecute más de un bloque de código por caso.
Sugiera una solución para prevenir que ocurra esto.

```
switch  (index) {
   case  1:
   case  3: odd += 1;
            sumodd += index;
   case  2:
   case  4: even += 1;
            sumeven += index;
   default: printf("Error in switch, index = %d\n", index);
}
```

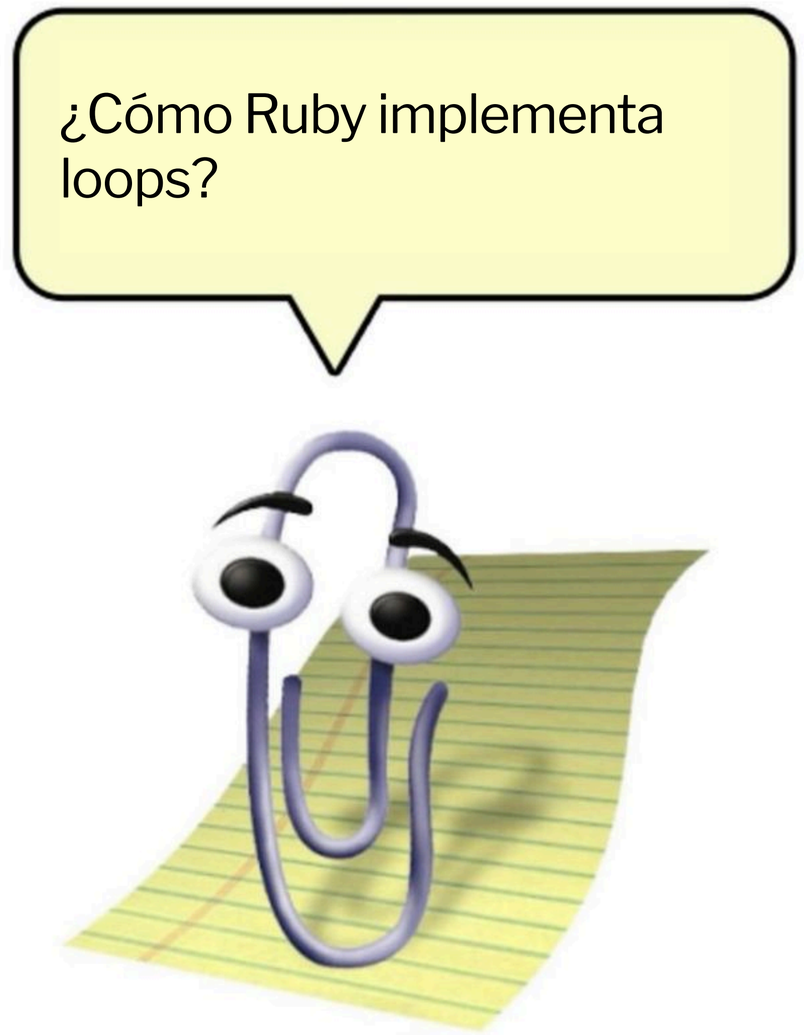
En el caso de los switch case, es posible que se ejecute más de un bloque de código por caso.
Sugiera una solución para prevenir que ocurra esto.

```
switch (value) {
    case -1:
        Negatives++;
        break;
    case 0:
        Zeros++;
        goto case 1;
    case 1:
        Positives++;
    default:
        Console.WriteLine("Error in switch \n");
}
```


¿Cómo Ruby implementa loops?

```
>> 4.times {puts "Hey!"}
Hey!
Hey!
Hey!
Hey!
=> 4
```

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```



¿Qué hace este código?

Y cuándo sería conveniente?

```
for x in mylist:
  if x.is_valid():
    break
  process_element(x)
else:
  process_error()
```

¿Qué hace este código?

Y cuándo sería conveniente?

Este es un ejemplo de syntatic sugar: el bloque debajo del *else* solamente se va a ejecutar si la lista se termina de iterar. Es decir, si el x válido es encontrado y escapamos del loop.
Esto es otro ejemplo de syntatic sugar.

```
for x in mylist:
  if x.is_valid():
    break
  process_element(x)
else:
  process_error()
```