

Estructuras de Datos

De C a C++

Referencias, clases, `std::string`, `std::vector` y pruebas

Gerson Garrido

Semana 1

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. De qué trata el curso
2. Lo que no cambia desde C, y namespaces
3. Referencias
4. De `struct` con funciones sueltas a clase
5. `std::string`, `std::vector`, y el error de copiar sin querer
6. Pruebas, y el laboratorio

> **Lectura**
>
> Apuntes de la semana 1. Apoyo: Open Data Structures, secs. 1.1 y 1.2. Código en `code/week-01`.

SECCIÓN 01

De qué trata el curso

01

La pregunta del curso

¿Cómo se organizan los datos para que un programa siga siendo rápido cuando los datos crecen?

Estructura de datos

Una forma de organizar datos en la memoria, junto con las operaciones que se pueden hacer sobre ellos.

- Vamos a **construir** las estructuras que las bibliotecas traen hechas.
- Vamos a **predecir** cuánto cuestan, y a **medirlas**.
- En C++: casi todo lo que saben de C sigue igual.

Buscar n códigos entre n estudiantes

lineal

```
algoritmo busqueda_lineal(codigos, n,
      codigo)
    para i desde 0 hasta n - 1 hacer
        si codigos[i] = codigo entonces
            devolver i
        fin si
    fin para
    devolver -1
fin algoritmo
```

binaria: exige la lista ordenada

```
algoritmo busqueda_binaria(codigos, n, codigo)
    bajo ← 0
    alto ← n - 1
    mientras bajo ≤ alto hacer
        medio ← bajo + (alto - bajo) div 2
        si codigos[medio] = codigo entonces
            devolver medio
        sino si codigos[medio] < codigo entonces
            bajo ← medio + 1
        sino
            alto ← medio - 1
        fin si
    fin mientras
    devolver -1
fin algoritmo
```

Organizar los datos de otra forma permite otro algoritmo.

Contar: da lo mismo en cualquier computadora

terminal

```
$ build/lookup 1000 linear
n = 1000, encontrados = 1000, comparaciones = 500500
$ build/lookup 1000 binary
n = 1000, encontrados = 1000, comparaciones = 8987
```

n	**Lineal**	**Binaria**
1 000	500 500	8 987
10 000	50 005 000	123 631
100 000	5 000 050 000	1 568 946
1 000 000	500 000 500 000	18 951 445
20 000 000	200 000 010 000 000	466 445 594

La lineal es exacta: mira $i + 1$ elementos para la posición i, y $1 + 2 + \cdots + n = n(n + 1)/2$.

Un segundo o un día

Búsqueda	n	Comparaciones	Tiempo
Lineal	40 000	800 020 000	0.291 s
Lineal	80 000	3 200 040 000	1.146 s
Binaria	10 000 000	223 222 809	0.514 s
Binaria	20 000 000	466 445 594	1.062 s

Mediana de 5 corridas con `-O2` y sin sanitizers. Los tiempos dependen de la máquina.

- Lineal: unas 2 800 millones de comparaciones por segundo en esta máquina.
- Con `n` = 20 000 000 serían 200 billones: casi 20 horas. Es una estimación: no la corrimos.
- Binaria, con la misma `n`: un segundo. Sin contar lo que cuesta ordenar.

Interfaz e implementación

Interfaz *(interface)*

Qué operaciones hay y qué hace cada una. `tally.h` era una.

Implementación *(implementation)*

Cómo se hace. El curso anterior vio dos para `Tally`.

Idea clave

Primero se pregunta qué operaciones necesita el programa y cuántas veces las hará. Después se elige la estructura. No hay una estructura mejor en general: la lista ordenada busca rápido, pero insertar en ella es caro.

Cómo está organizado el curso

- **Cuatro partes:** de C a C++ y costos; secuencias y ordenamiento; diccionarios; prioridades y grafos.
- **Cada semana:** 2 h de teoría y dos laboratorios de 2 h, con un programa real que necesita la estructura.
- **Cuatro prácticas**, en las semanas 4, 7, 11 y 14, un examen parcial en la semana 8 y uno final en la 16.
- **Los apuntes son el texto del curso.** Open Data Structures es lectura de apoyo.

SECCIÓN 02

Lo que no cambia desde C

02

Esto ya es C++

c_style.cpp

```cpp
#include <cstdio>

struct point {
    int x;
    int y;
};

int manhattan(const point *p) {
    int dx = p->x < 0 ? -p->x : p->x;
    int dy = p->y < 0 ? -p->y : p->y;
    return dx + dy;
}
```

terminal

```
$ build/c_style
0
7
```

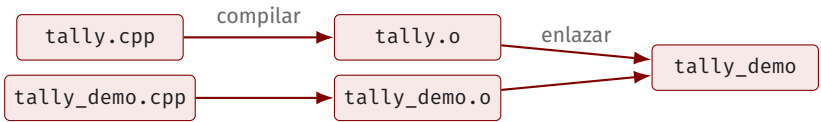
No cambia: tipos, control, funciones, punteros, `struct`, encabezados, memoria, ownership, `make`, `gdb`, valgrind, sanitizers.

Cambia: `g++` en vez de `gcc`, y `-std=c++20`.

Compilar y enlazar es igual

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined -c tally.cpp -o build/tally.o
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined -c tally_demo.cpp -o build/tally_demo.o
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined build/tally.o build/tally_demo.o -o build/tally_demo
$ build/tally_demo
3 valores, suma 40, mayor 17
```



Diferencias pequeñas

En C	En C++	Por qué importa
`#include <stdio.h>`	`#include <cstdio>`	Sus nombres están en `std::`
`int` o `stdbool.h`	`bool`, `true`, `false`	Son parte del lenguaje
`NULL`	`nullptr`	El único puntero nulo que usamos
`struct point p;`	`point p;`	El nombre ya es un tipo
`int main(void)`	`int main()`	Vacío significa “sin parámetros”
`(double) x`	`static_cast<double>(x)`	Explícita, y fácil de buscar
`void *` se convierte solo	Hay que convertirlo	C++ es más estricto

`malloc` sin conversión: `error: invalid conversion from 'void*' to 'int*'`. En este curso no se usa `malloc`. También hay palabras reservadas nuevas: una variable no se puede llamar `class` ni `new`.

std::cout

hello.cpp

```cpp
#include <iostream>

int main() {
    int students = 32;
    std::cout << "Hola, Estructuras de Datos\n";
    std::cout << "Somos " << students << " estudiantes
      \n";
    return 0;
}
```

terminal

```
$ build/hello
Hola, Estructuras de Datos
Somos 32 estudiantes
```

- « manda un valor a la salida. Se puede encadenar.
- Sin `%d`: el compilador conoce el tipo de cada valor.
- `std::cerr` es `stderr`. `std::cin » x` lee, y `while (std::cin » x)` sigue mientras haya algo.

Namespaces: un prefijo, hecho parte del lenguaje

namespaces.cpp

```cpp
namespace circle {
double area(double radius) {
    return 3.14159 * radius * radius;
}
}  // namespace circle

namespace square {
double area(double side) {
    return side * side;
}
}  // namespace square

int main() {
    std::cout << circle::area(2.0) << "\n";
    std::cout << square::area(2.0) << "\n";
```

terminal

```
$ build/namespaces
12.5664
4
```

Namespace

Un alcance con nombre. Evita que dos nombres choquen sin usar prefijos como `tally_add`.

La biblioteca estándar está en `std`.

Por qué no `using namespace std`

errors/using_std_collision.cpp

```cpp
using namespace std;

int count = 0;
...
        count += value;
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/using_std_collision.cpp -o build/using_std_collision
errors/using_std_collision.cpp:13:9: error: reference to 'count' is ambiguous
...
errors/using_std_collision.cpp:8:5: note:                 'int count'
```

Regla del curso: siempre `std::`. Trae miles de nombres que no conocen, puede romperse con una versión nueva de la biblioteca, y en un encabezado se contagia.

SECCIÓN 03

Referencias

03

Intercambiar, de dos formas

swap_both.cpp

```cpp
void swap_pointer(int *a, int *b) {
    int saved = *a;
    *a = *b;
    *b = saved;
}
```

Se llama con `swap_pointer(&x, &y)`

swap_both.cpp

```cpp
void swap_reference(int& a, int& b) {
    int saved = a;
    a = b;
    b = saved;
}
```

Se llama con `swap_reference(x, y)`

Referencia *(reference)*

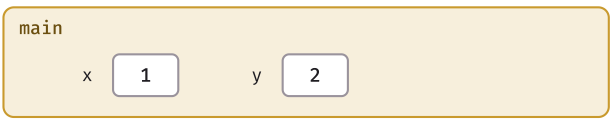
Otro nombre para un objeto que ya existe. `int& alias = grade;` no crea otro entero.

`&` en un **tipo** declara una referencia. `&` en una **expresión** sigue siendo la dirección, como en C.

Otro nombre en la misma casilla

CÓDIGO

```
swap_reference(x, y);
int saved = a;
a = b;
b = saved;
}
```



Al retornar, el marco desaparece con saved y los dos nombres. El cambio queda en main

Puntero y referencia

	Puntero `int *p`	**Referencia `int& r`**
¿Es un objeto con su propia dirección?	Sí	No: `&r` da la del objeto
¿Debe iniciarse al declararse?	No	Sí
¿Puede ser nulo?	Sí, `nullptr`	No en un programa válido
¿Puede cambiar a qué se refiere?	Sí, `p = &y`	No: `r = y` asigna al objeto
¿Tiene aritmética?	Sí, `p + 1`	No
¿Cómo se usa?	`*p`, `p->campo`	`r`, `r.campo`

Internamente, un compilador suele implementar una referencia como una dirección. Por eso se parecen tanto.

`alias = other` no cambia de objeto

reference_basics.cpp

```cpp
int grade = 14;
int& alias = grade;

alias = 15;
std::cout << "grade = " << grade << "\n";

// Esto no cambia a quién se refiere alias: copia 20
    dentro de grade.
int other = 20;
alias = other;
other = 0;
```

terminal

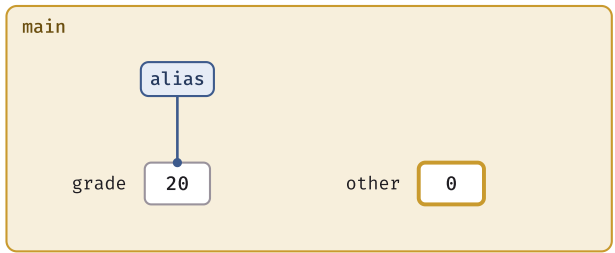
```
$ build/reference_basics
grade = 15
misma dirección: 1
grade = 20, other = 0
```

Y debe iniciarse al declararse. Sin eso, `g++` dice: `error: 'alias' declared as reference but not initialized`

La etiqueta no se mueve

CÓDIGO

```
int grade = 14;
int& alias = grade;
alias = 15;
int other = 20;
alias = other;
other = 0;
```



Cambiar other no toca a grade. El programa imprime grade = 20, other = 0

No puede ser nula, pero sí quedar colgando

errors/return_local_ref.cpp

```cpp
int& next_code(int code) {
    int result = code + 1;
    return result;
}
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/return_local_ref.cpp -o build/return_local_ref
errors/return_local_ref.cpp:6:12: warning: reference to local variable 'result' returned [-Wreturn-local-addr]
```

- Una función que recibe `int&` no tiene que revisar si es nula.
- `int& r = *p` con `p` nulo no crea una “referencia nula”: seguir ese puntero ya es comportamiento indefinido.
- Si el objeto muere primero, queda una referencia colgante. A una local, nunca: devuelvan por valor.

Referencias constantes

const_forms.cpp

```cpp
const int *to_const = &a;    // no se puede escribir *to_const
to_const = &b;               // sí se puede cambiar el puntero
int *const fixed = &a;       // no se puede cambiar el puntero
*fixed = 5;                  // sí se puede escribir a través de él
const int& view = a;         // no se puede escribir a través de view
a = 7;                       // a sí cambia, y view lo ve
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/const_ref_write.cpp -o build/const_ref_write
errors/const_ref_write.cpp:3:11: error: assignment of read-only reference 'value'
```

Se lee de derecha a izquierda. `const` es una promesa de esa vista, no del objeto: el objeto puede cambiar por otro camino, y la referencia lo verá.

`const&` se puede atar a una temporal

const_forms.cpp

```cpp
std::size_t count_letter(const std::string& text, char letter) {
...
std::cout << count_letter("banana", 'a') << "\n";
```

- El compilador crea una `std::string` temporal con `"banana"`, y `text` se ata a ella.
- La temporal vive hasta que termina la expresión de la llamada.

errors/ref_to_temporary.cpp

```cpp
std::size_t length(std::string& text) {
```

Sin `const`, no compila, y con razón: la función podría modificar una temporal que nadie verá.

```
error: cannot bind non-const lvalue reference of type 'std::string&' ... to an rvalue of type 'std::string'
```

Cómo pasar un argumento

Qué necesita la función	Cómo	Ejemplo
Leer un valor pequeño	Por valor	`int grade`
Leer algo que puede ser grande	Por `const&`	`const std::string& name`
Modificar lo que le pasan	Por `&`	`std::vector<int>& values`
Quedarse con su copia para cambiarla	Por valor	`std::string text`
Recibir algo que puede faltar	Por puntero	`const Node *node`

Por defecto

Los tipos pequeños, por valor. Lo demás, `const&`. Si van a modificar el argumento, `&`.

SECCIÓN 04

De `struct` a clase

04



El módulo `Tally` en C: lo que costaba

tally_c/tally_demo.c

```c
Tally *tally = tally_new();
if (tally == NULL) {
    return 1;
}
int ok = tally_add(tally, 14) && tally_add(tally, 9) && tally_add(tally, 17);
if (!ok) {
    tally_free(tally);
    return 1;
}
printf("%d valores, suma %d, mayor %d\n", tally_count(tally), tally_sum(tally), tally_max(tally))
    ;
tally_free(tally);
```

- Un prefijo en cada función, y `tally` como primer argumento en cada llamada.
- Revisar `NULL` y acordarse de liberar en cada salida.
- Esconder los campos obliga a usar el heap: afuera nadie sabe cuánto mide.

El mismo módulo, como clase

tally.h

```
1  class Tally {
2  public:
3      void add(int value);
4      std::size_t count() const;
5      int sum() const;
6      int max() const;
7  private:
8      std::vector<int> values_;
9  };
```

tally_demo.cpp

```
1  Tally tally;
2  tally.add(14);
3  tally.add(9);
4  tally.add(17);
```

terminal

```
$ build/tally_c_demo
3 valores, suma 40, mayor 17
$ build/tally_demo
3 valores, suma 40, mayor 17
```

Sin puntero, sin revisar memoria, y sin liberar.

Métodos

Clase *(class)*

Tipo que junta **campos** con los **métodos** que trabajan sobre ellos. Un valor de ese tipo es un **objeto**.

C	C++
`int tally_add(Tally *tally, int value)`	`void Tally::add(int value)`
`tally_add(tally, 14);`	`tally.add(14);`
`tally->values` adentro de la función	`values_` adentro del método

Un método recibe un parámetro oculto, `this`: un puntero al objeto. `tally.add(14)` funciona como si fuera `Tally::add(&tally, 14)`.

Los campos terminan en _, para distinguirlos de las variables locales.

Público y privado

errors/private_access.cpp

```
1  tally.values_.clear();
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/private_access.cpp -o build/private_access
errors/private_access.cpp:6:11: error: 'std::vector<int> Tally::values_' is private within this context
errors/../tally.h:23:22: note: declared private here
```

- `class` y `struct` son lo mismo, salvo el acceso por defecto: en `class`, privado; en `struct`, público.
- Es **por clase, no por objeto**: un método puede ver los campos privados de otro objeto de la misma clase.
- Convención: `class` si hay algo que proteger; `struct` si solo agrupa datos a la vista.

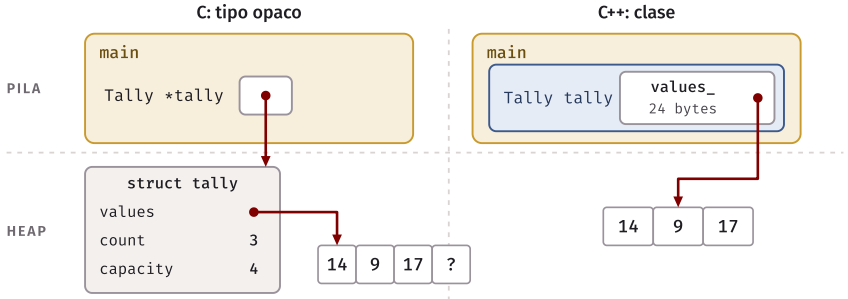
`private` y el tipo opaco

	Tipo opaco en C	Clase con `private`
¿Quien usa el tipo ve los campos?	No	Sí, en el encabezado
¿Puede usarlos?	No: no los conoce	No: lo prohíbe el compilador
¿El compilador sabe su tamaño afuera?	No	Sí
¿Hay variables locales del tipo?	No, solo punteros	Sí
¿Dónde vive el objeto?	En la práctica, en el heap	Donde quien lo usa decida
Al cambiar campos, se recompila	Solo el `.c` del módulo	Todo lo que incluye el `.h`

Respuesta

`private` reemplaza la garantía de que nadie afuera toca los campos. Es control de acceso, no ocultamiento.

Dónde está cada cosa



Los 24 bytes dependen del sistema. Los enteros siguen en el heap, y su owner es el `std::vector`. La semana 2 explica cómo libera su bloque.

Constructores

course.cpp

```cpp
class Course {
public:
    Course(const std::string& name, int credits) :
      name_(name), credits_(credits) {
        assert(credits > 0);
    }
...
private:
    std::string name_;
    int credits_;
    int enrolled_ = 0;
};
```

Constructor *(constructor)*

Método sin tipo de retorno y con el nombre de la clase. Corre al crear el objeto.

- La **lista de inicialización** inicia cada campo antes de entrar a las llaves.
- Sin constructores, el compilador genera uno. Un campo `int` sin valor por defecto queda con basura.

Tres errores con constructores

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/no_default_ctor.cpp -o build/no_default_ctor
errors/no_default_ctor.cpp:16:11: error: no matching function for call to 'Range::Range()'
```

Al declarar cualquier constructor, el compilador ya no genera el que no recibe nada.

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/init_order.cpp -o build/init_order
errors/init_order.cpp:14:9: warning: 'Range::high_' will be initialized after [-Wreorder]
```

Los campos se inician en el orden en que están **declarados**, no en el de la lista. `Tally tally();` declara una **función**, y `g++` dice: `non-class type 'Tally()'`. El objeto: `Tally tally;`

Métodos `const`

> **Método `const` *(const method)***
>
> Promete no modificar el objeto: adentro, `this` es un puntero a un objeto constante.

1. No puede modificar un campo, ni llamar a un método que no sea `const`.
2. Sobre un objeto `const` o un `const&`, solo se pueden llamar métodos `const`.

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/const_method_writes.cpp -o build/const_method_writes
errors/const_method_writes.cpp:5:9: error: increment of member 'Counter::value_' in read-only object
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/const_object_call.cpp -o build/const_object_call
errors/const_object_call.cpp:5:14: error: passing 'const Tally' as 'this' argument discards qualifiers [-fpermissive]
```

Regla: todo método que no modifica el objeto se marca `const` desde el principio.

Lo que `const` no garantiza

const_shallow.cpp

```cpp
// Compila: no cambia target_, solo el entero
     al que apunta.
void reset() const {
    *target_ = 0;
}
```

terminal

```
$ build/const_shallow
0
```

- Protege los **campos** del objeto, no lo que hay al otro lado de un puntero.
- No promete que el objeto no cambie: otra parte del programa puede tenerlo por una referencia sin `const`.
- Salidas que no usamos: campos `mutable` y `const_cast`.

En el `.h` la clase, en el `.cpp` los métodos

tally.cpp

```cpp
void Tally::add(int value) {
    values_.push_back(value);
}

std::size_t Tally::count() const {
    return values_.size();
}

int Tally::sum() const {
    int total = 0;
    for (int value : values_) {
        total += value;
    }
    return total;
}
```

`Tally::` dice de qué clase es. El `const` va también aquí: es parte de la firma.

Qué desapareció

	C	C++
Líneas de la implementación	68	30
Líneas del encabezado	27	26
Líneas del programa que lo usa	19	13
Llamadas a `malloc`, `realloc` y `free` en el módulo	4	0
Chequeos de memoria al usarlo	2	0

- La memoria sigue ahí, en un bloque del heap. Su owner ahora es el `std::vector`.
- Si no hay memoria, `push_back` lanza una excepción, que sin atrapar termina el programa.

SECCIÓN 05

std::string, std::vector y copias

05



`std::string`: owner de sus caracteres

strings.cpp

```cpp
std::string first = "Ana";
std::string full = first + " Quispe";
std::string copy = full;
copy[0] = 'E';

std::cout << full << " / " << copy << "\n";
std::cout << "largo: " << full.size() << "\n";
std::cout << "iguales: " << (full == copy) << "\n";

std::size_t space = full.find(' ');
std::cout << "apellido: " << full.substr(space + 1)
          << "\n";
```

terminal

```
$ build/strings
Ana Quispe / Ena Quispe
largo: 10
iguales: 0
apellido: Quispe
```

`strlen` → `s.size()`
`strcpy` → `a = b`
`strcat` → `a += b`
`strcmp` → `a == b`, `a < b`
`strstr` → `s.find(...)`

Copiar es copiar los caracteres, y `==` compara el contenido. Sin tamaños y sin `free`. Para una función de C: `s.c_str()`.

`std::vector`: un arreglo que crece

vectors.cpp

```cpp
std::vector<int> grades;
grades.push_back(14);
grades.push_back(9);
grades.push_back(17);
std::cout << "tamaño: " << grades.size() << "\n";
std::cout << "último: " << grades.back() << "\n";

grades[1] = 11;
grades.erase(grades.begin());
std::cout << "primero tras borrar: " << grades[0]
          << "\n";
std::vector<int> primes = {2, 3, 5};
std::vector<int> other = primes;
other[0] = 7;
```

terminal

```
$ build/vectors
tamaño: 3
último: 17
primero tras borrar: 11
2 7
```

- `<int>` dice el tipo de los elementos. Es un **template**: semana 2.
- Copiarlo copia todos sus elementos.
- `erase` corre los siguientes: borrar al inicio es caro.

El tamaño no tiene signo

`for (int i = 0; i < values.size(); i++)` da una advertencia: `[-Wsign-compare]`. Y restar puede dar la vuelta:

errors/size_minus_one.cpp

```cpp
std::vector<int> empty;
std::cout << empty.size() - 1 << "\n";
```

terminal

```
$ build/size_minus_one
18446744073709551615
```

En los enteros sin signo esto está definido: dan la vuelta, y UBSan no se queja. Usen `for` por rango, o `std::size_t i` con `i + 1 < v.size()`.

El índice no se revisa

terminal

```
$ build/vector_index
READ of size 4 at 0x50200000001c thread T0
    #0 0xaaaaba1753cc in main errors/vector_index.cpp:7
0x50200000001c is located 0 bytes after 12-byte region [0x502000000010,0x50200000001c)
```

`values[3]` con tres elementos: fuera del bloque, ASan lo ve.

terminal

```
$ build/vector_reserved
size = 1, values[5] = 42
```

Con `reserve(8)`: fuera de `size()` y dentro del bloque. Es un error, y ASan no lo ve.

`values.at(3)` revisa contra `size()`, y termina: `terminate called after throwing an instance of 'std::out_of_range'`

`for` por rango

range_for.cpp

```cpp
// x es una copia de cada elemento.
for (int x : grades) {
    x = x + 1;
    std::cout << x << " ";
}
std::cout << "/ " << grades[0] << " " << grades
     [1] << "\n";

// x es otro nombre del elemento.
for (int& x : grades) {
    x = x + 1;
}
std::cout << grades[0] << " " << grades[1] << "\n
     ";
```

terminal

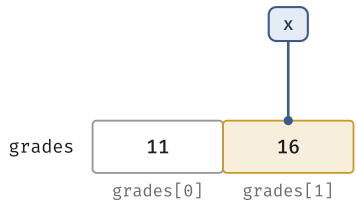
```
$ build/range_for
11 16 / 10 15
11 16
```

`int x`	copia, tipos pequeños
`int& x`	para modificar
`const T& x`	leer tipos grandes

Es la misma regla de los parámetros.

`int x` e `int& x`, vuelta por vuelta

```
for (int& x : grades) {  x = x + 1;  }
```



En la segunda vuelta, x es otro nombre de grades[1]. El programa imprime 11 16

`auto` solo es una copia

range_for.cpp

```cpp
// auto deduce el tipo; sin & sigue siendo una
    copia.
for (auto name : names) {
    name += "!";
}
std::cout << names[0] << "\n";
for (auto& name : names) {
    name += "!";
}
std::cout << names[0] << "\n";
```

terminal

```
$ build/range_for
Ana
Ana!
```

Error común: `for (auto x : v)` copia cada elemento, sin aviso. Por defecto, `const auto&`; para modificar, `auto&`.

Pasar un vector por valor copia todo

copy_or_not.cpp

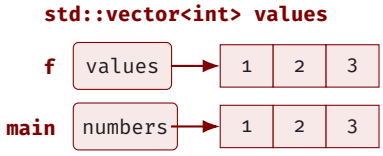
```cpp
void by_value(std::vector<int> values, const int *original) {
...
void by_const_reference(const std::vector<int>& values, const int *original) {
```

terminal

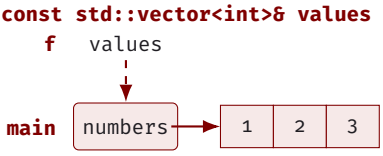
```
$ build/copy_or_not
por valor:      mismo bloque = 0, values[0] = 99
en main:        numbers[0] = 1
por const&:     mismo bloque = 1, values[0] = 1
```

- **Por valor:** otro bloque, y una copia de cada elemento. **Por `const&`:** otro nombre, y ninguna copia.
- Compila sin avisar y funciona bien. Solo es lenta, y más mientras más datos haya.

Por valor y por `const&`, dibujado



dos bloques



un bloque, y otro nombre

Respuesta

Para leer un arreglo grande conviene `const&`: hace lo mismo y no copia nada. Por valor, se paga copiar `n` elementos al entrar, y liberarlos al salir.

Dentro de un ciclo: de lineal a cuadrático

value_at.cpp

```cpp
int value_at_copy(std::vector<int> values, std
    ::size_t index) {
...
int value_at_ref(const std::vector<int>&
    values, std::size_t index) {
```

terminal

```
$ build/copy_in_loop 1000 copy
suma = 1000, elementos copiados = 1000000
$ build/copy_in_loop 1000 ref
suma = 1000, elementos copiados = 0
```

n	Copia	const&
20 000	0.047 s	0.001 s
40 000	0.183 s	0.002 s
80 000	0.742 s	0.002 s

Mediana de 5 corridas, `-O2`, sin sanitizers. Dependen de la máquina; bajo unos ms, se mide sobre todo el inicio.

Al duplicar `n`, la copia tarda casi 4 veces más: `n` llamadas que copian `n` elementos. Un `const&` puede convertir un programa cuadrático en lineal.

SECCIÓN 06

Pruebas y laboratorio

06

assert

Si la condición es falsa, imprime archivo, línea y condición, y detiene el programa. Con `-DNDEBUG`, desaparece, y con él lo que hace adentro. `-O2` no lo define.

errors/assert_side_effect.cpp

```cpp
values.pop_back();
assert((values.push_back(9), values.size() == 3));
std::cout << values.size() << "\n";
```

terminal

```
$ build/assert_side_effect
3
$ build/assert_off
2
```

Regla: lo que está adentro de un `assert` no debe hacer nada necesario para el programa.

Casos dirigidos y operaciones al azar

tally_tests.cpp

```cpp
void test_against_reference() {
    std::mt19937 random(2026);
    Tally tally;
    std::vector<int> reference;
    for (int step = 0; step < 2000; step++) {
        int value = static_cast<int>(random() % 201) - 100;
        tally.add(value);
        reference.push_back(value);
        assert(tally.count() == reference.size());
        assert(tally.sum() == std::accumulate(reference.begin(), reference.end(), 0));
```

- **Dirigidos:** cada comportamiento y cada caso borde. También a través de un `const&`.
- **Al azar:** con semilla fija, contra algo en lo que confían. Si falla, falla igual la próxima vez.
- Siempre con `-Wall -Wextra` y sanitizers, sin advertencias.

Laboratorio: portar un módulo de C a C++

Sesión 1

1. `env/check.sh`, y un programa corto con el `Makefile` dado.
2. Leer `lab/c/`: un registro de notas, con tipo opaco.
3. Portarlo a la clase `GradeBook`: `std::string`, `std::vector` y métodos.

Sesión 2

4. Pasar las pruebas a la clase, y agregar casos borde.
5. Agregar `average` y `best`, `const`. Ver que el compilador rechaza un método `const` que modifica un campo.
6. Comparar las dos versiones.

Comandos

`make lab`, `make test-lab` y `make compare`. Una prueba corre órdenes al azar en la versión en C y en la suya, y compara lo que imprimen.

Para la próxima clase

- Terminen el laboratorio: `make test-lab` debe pasar.
- Sepan responder sin mirar:
  1. ¿Qué diferencia hay entre `void f(std::vector<int> v)` y `void f(const std::vector<int>& v)`? ¿Cuál conviene para leer un arreglo grande?
  2. ¿Puede una referencia quedar sin apuntar a nada, como un puntero nulo? ¿Puede cambiar a qué se refiere?
  3. ¿Qué garantiza un método marcado `const`?
  4. ¿Qué parte del tipo opaco de C reemplaza `private`?

Semana 2

Ownership y RAII: cómo logra `std::vector` liberar su memoria solo. Y escribiremos una clase que hace lo mismo.