Compilando Conocimiento

Introducción a lo que tienes que saber sobre Programación Competitiva usando C++

Club de Algoritmia ESCOM

Rosas Hernandez Oscar Andrés

Enero 2018

Índice general

Parte I

Introducción a la Programación Competitiva

Capítulo 1

¿Qué es la Programación Competitiva?

Figura 1.1: Imágen por Sharaft Siddiqui Reheb

1.1. Introducción

‘‘ Given well-known CS (computer science) problems,
solve them as quickly as possible! ’’

‘‘ Dados problemas famosos de ciencias de la computación,
¡resuelvelos tan rápido como puedas! ’’

- Competitive Programming 3 [1]

La programación competitiva es la actividad de resolver *problemas bastante conocidos de ciencias de la computación* mediante la creación de *programas* que obtengan la respuesta dentro de un cierto *límite*.

- **¿Problemas conocidos de ciencias de la computación?**

  Los problemas que vamos a resolver están bien definidos, es decir son problemas en los que para cualquier entrada tu tendrías que ser capaz de calcular la salida a mano.

  Además estarás informado de todas las restricciones del problema y todas las suposiciones que puedes tomar para facilitarte la vida.

- **Tendrás que programar.**

  Si (pero no como en tú día a día, no harás una aplicación web o con una interfaz super bonita), sino que haras programas que toman datos por la entrada estándar y nos regresan una respuesta por la salida estándar, es decir, un programa de terminal. Esto porque lo más importante aquí es el algoritmo.

- **Tendrás que cumplir límites.**

  Tu programa tendrá que resolver el problema con unas restricciones en tiempo y memoria, por ejemplo, tu programa tendrá que dar la solución en menos de 300ms y ocupando menos de 300MB de memoria.

1.1.1. Ejemplos

- On ta el pinche fácil pa irme? - OmegaUp
- Factores comunes - OmegaUp
- Two Sum II: Input array is sorted - Leetcode

1.2. Propiedades de un problema

- **Son calificados por una máquina.**

  Generalmente usamos onlines judges (jueces en línea) para poder saber si hemos resuelto un problema, es decir, al final del día nuestro problema lo califica un programa.

  Así que no hay puntos medios o tu programa funciona siempre y como debe o el juez te dirá que está mal.

Time Submitted	Status	Runtime	Memory	Language
14 days ago	Accepted	4 ms	9.6 MB	cpp
17 days ago	Accepted	8 ms	9.7 MB	cpp
17 days ago	Wrong Answer	N/A	N/A	cpp
17 days ago	Wrong Answer	N/A	N/A	cpp
17 days ago	Wrong Answer	N/A	N/A	cpp
17 days ago	Runtime Error	N/A	N/A	cpp
17 days ago	Runtime Error	N/A	N/A	cpp
17 days ago	Runtime Error	N/A	N/A	cpp

- **Tienen historia.**

  Muchas veces (casi siempre) los problemas están metidos dentro de una historia, está ayuda a que el problema sea mucho más interesante y que te cueste más entender de que trata, que es lo que en fondo te piden resolver.

  Además, personalmente, ayuda mucho a la hora de aprender, pues es mucho más fácil que recuerdes una técnica por un problema en especial (por ejemplo, que recuerdes el problema de la mochila) que te gusto mucho en vez de que recuerdes temas matemáticos o de computación puros y duros.

Descripción

Richi es un gran fan de las competencias de programación y ha estudiado bastante para el 8vo concurso anual de programación, desafortunadamente pocos días antes de éste, se enteró que debe entregar un proyecto exactamente en la fecha del evento, como richi es muy rebelde decide llegar unos minutos después a su clase y antes de ello asistir al concurso para resolver el problema más fácil de la competencia.

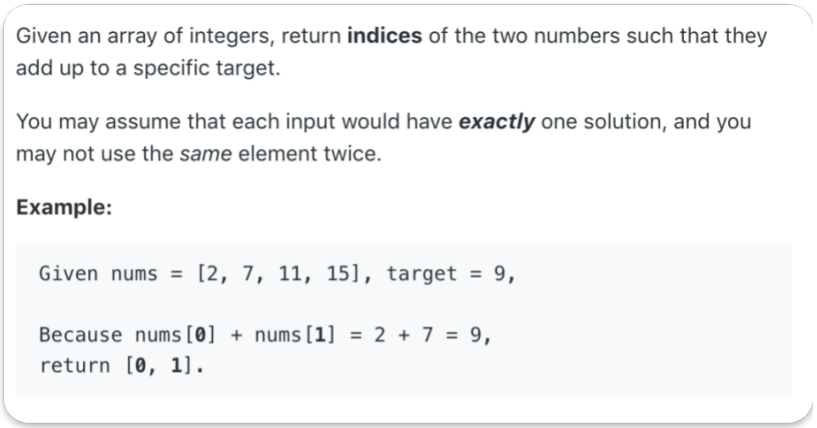
El día llegó y rápidamente richi el rebelde encontró entre todos los problemas uno que le pareció demasiado fácil que consistía en lo siguiente:

Tienes un tablero de NxM casillas, quieres enumerar cada casilla con números consecutivos de izquierda a derecha y de arriba abajo, comenzando en la esquina superior izquierda con el número 1, dime ¿Cuál será el último número que escribirás en el tablero?

- **Te dan ejemplos.**

  Incluso aunque el problema te dice que es exactamente lo que te está pidiendo que resuelvas es mucho más fácil para los humanos entenderlos si nos dan ejemplos.

  Esto también ayuda a que no malinterpretemos el problema y empezemos a resolver algo que no era lo que teníamos que hacer.



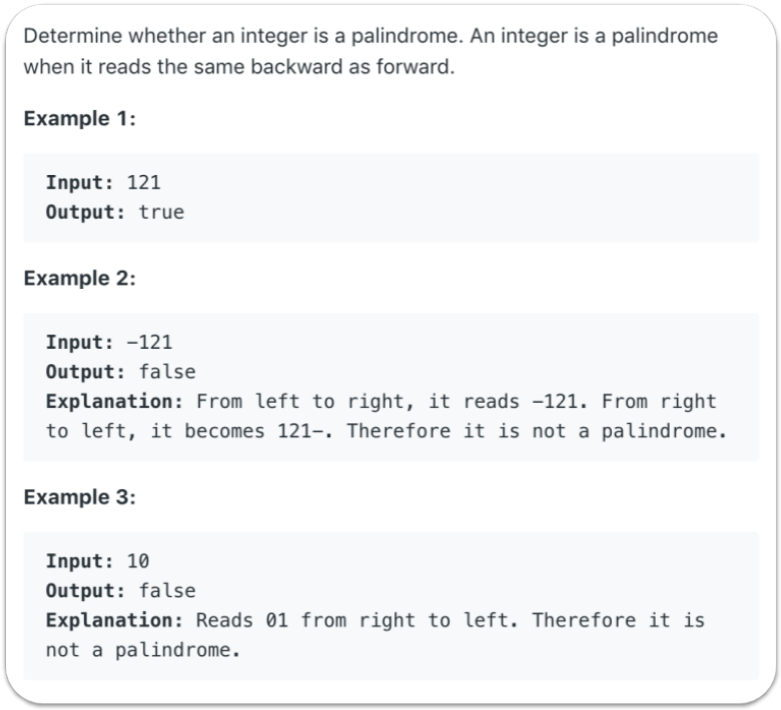
Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```



Determine whether an integer is a palindrome. An integer is a palindrome when it reads the same backward as forward.

Example 1:

```
Input: 121
Output: true
```

Example 2:

```
Input: -121
Output: false
Explanation: From left to right, it reads -121. From right
to left, it becomes 121-. Therefore it is not a palindrome.
```

Example 3:

```
Input: 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is
not a palindrome.
```

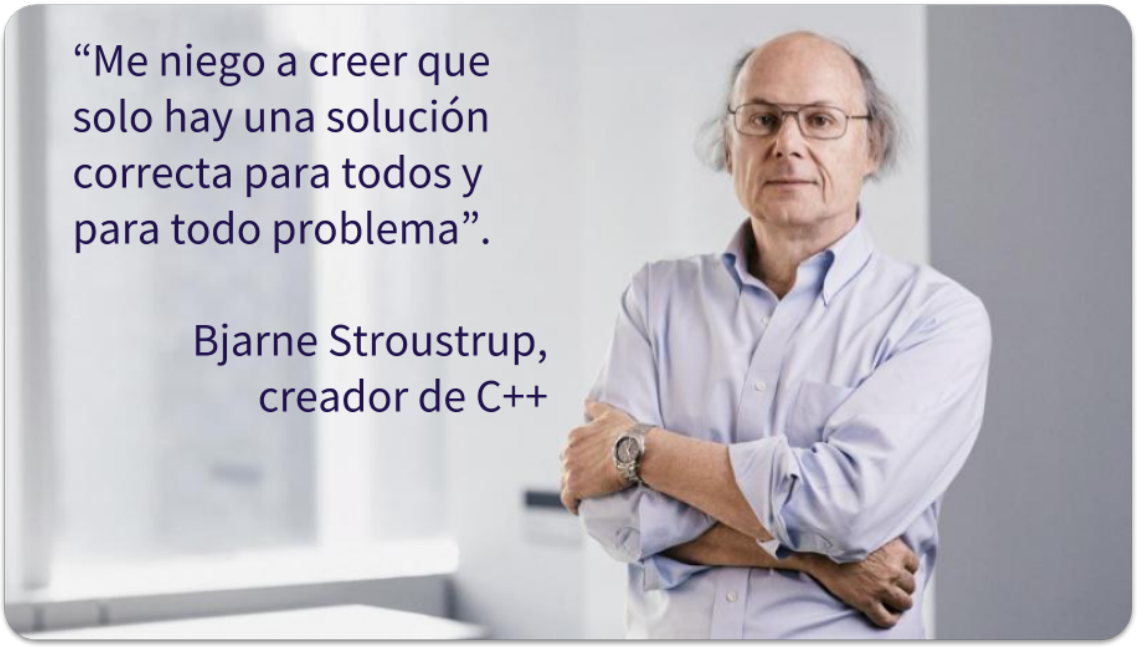
- **El corazón del problema está relacionado siempre con matemáticas, lógica o ciencias de la computación.**

Parte II

Un tour de C++ y lo que deberías saber de el

Capítulo 2

¿Porqué usar C++ ?



2.1. Grandes ventajas de C++

Hay muchas razones por las cuales C++ es uno de los lenguajes más usados en la modernidad en la programación competitiva (si no es que el más).

Puedes escuchar un video muy bonito para mi sobre porque elegir este lenguaje:

`Bjarne Stroustrup: Why I Created C++`

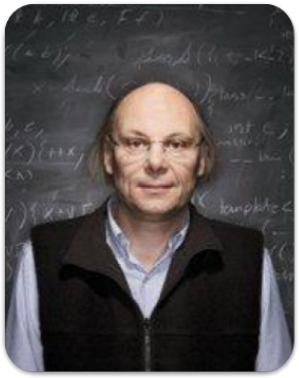


Figura 2.1: Este es el creador del lenguaje (reto: decir su nombre en voz alta) Bjarne Stroustrup

Las principales razones por C++ sin un orden en particular son:

- **Te da abstracciones sin costo extra.**

  O como lo diría su creador te da el poder de usar abstracciones de alto nivel pero sin que las mismas hagan tu programa mas innecesariamente mas lento, o como lo dice la comunidad en inglés:

  > " C++ has the zero-overhead principle: `what you don't use, you don't pay for` , that is, with every new feature added to the language, you get at least as good performance as if that feature will not be included.
  >
  > Also there are the principle: `What you do use cost either only as much as what you'd implement yourself, or it cost less` , meaning that a new feature can either mantain or improve the performance. "

  Personalmente creo que esa es mayor ventaja que tiene sobre todos los demás y que resume perfectamente todo el objetivo de C++ .

  Lo que nos dice esto es que podremos representar grafos, matrices, operaciones, clases en general, algoritmos, etc... en nuestros programas con gran facilidad, cosa que no podemos hacer fácilmente por ejemplo en lenguajes como C o la familia de los ensambladores.

Y podrías decir, pero en Java o en Python podemos representar de una manera igual de fácil así que ... ¿porqué usar C++ ?

Pues porque en todos los demás lenguajes estas pagando un costo (a veces muy grande de varios cientos o miles de veces) por poder representar ideas o conceptos abstractos en un programa, en C++ esta prácticamente garantizado que usar una clase por ejemplo o que usar un arreglo que se auto dimensiona (vector) no será más costoso que si lo hubieras hecho tu desde ensamblador.

- **Tienes a la biblioteca estándar, la gran std::* a tu lado**

  Esta es otra gran ventaja, ya que siguiendo con la mentalidad de abstracciones sin costo extra tenemos gracias a la gran libreria estándar un montón de cosas que no tenemos que hacer desde cero, desde arreglos que cambian de tamaño, arreglos asociativos, pilas, colas, algoritmos de ordenamiento, de búsqueda, algoritmos para filtrar información, para hacer particiones o permutaciones, etc...

  Así podemos dejar los algoritmos básicos al lenguaje y enfocarnos en las cosas que son de verdad interesantes.

- **Es "statically typed and compiled"; el compilador es tu amigo**

  Otra ventaja más, podemos siempre confiar en el compilador, en que si nuestro programa compila muy probablemente está haciendo lo que debe hacer (cosa que no podemos esperar con python por ejemplo), el compilador es tu amigo, te dira en donde te equivocaste, en donde puede que hayas querido decir otra cosa y muchas veces te dará consejos.

  Además, tras bambalinas está transformando tu código en algo que la computadora puede de verdad entender y además usará toda la información que le diste para muchas veces incluso mejorar tu código en vez de solo "traducirlo" y optimizarlo de maneras que me sorprenden personalmente.

- **Es prácticamente un "super set" de C**

  Es decir, que (casi) cualquier código válido de C es válido en C++ , esto es de gran ayuda pues C es uno de los lenguajes más conocidos por lo que puede que la sintaxis de C++ sea más fácil que entender la sintaxis de Haskell, de Kotlin o Prolog, por ejemplo.

  Otra ventaja es que al estar basado en C conserva muchas de las ventajas de C como su portabilidad, su velocidad de ejecución y la capacidad de tener un gran control de todos los recursos del sistema (memoria y tiempo de vida de un objecto, cough cough Java y su recolector de basura)

- **Tienes un gran control de todos los recursos del sistema**

  Esto es también es muy importante, pues nos dice que en C++ podemos controlar con gran lujo de detalle los recursos del sistema, como por ejemplo la memoria.

  C++ nos da el control de decidir por ejemplo a que lugar va cada variable (heap o al stack), lo cual es esencial para hacer un programa de alto rendimiento (y creeme que lo necesitaras).

  Es determinista.

  Es decir la limpieza de los objectos una vez que ya se acabaron de usar es solicitada cuanto tu quieres, y no *cuando el recolector de basura decide hacerlo.*

  Tenemos el control de decidir si queremos pasar las cosas por referencia o por valor, si deseamos mover un objecto o si una referencia no podrá ser modificada.

- **Tienes *value types* por defecto (pero también tiene referencias si las necesitas)**.

  Esta quiza sea algo rara de explicar si es que no conoces muchos sobre lenguajes de computación, y si no la entiendes no te preocupes.

  Lo que nos dice es que las variables en C++ son variables de valor por defecto, es decir cuando decimos `auto x = 10` nuestra variable `x` de verdad es el fragmento de memoria que tiene ese 10.

  O cuando decimos `vector<int> vec` nuestra variable `vec` de verdad tiene sus elementos adyacentes en memoria, y no como en otros lenguajes que lo que es adyacente es un montón de punteros.

  En otras palabras que nuestras variable de verdad almacenan la información que queremos **y no una referencia que apunta a donde esta nuestra información** .

  Claro que aún puedes expresar la idea de las referencias, pero por defecto hablamos de variables que almacenan valores.

2.2. C++ vs otros lenguajes

2.2.1. vs C

El gran problema con C es que es un lenguaje muy pequeño en el sentido en que todo lo tienes que hacer tu, si quieres hacer un problema que involucra cosas medio complejas todas las estructuras las tienes que codear al momento, y en un deporte de tiempo, cada segundo cuenta, así que en resumén, lo que "mata" a C es la falta de algo parecido a la std de C++ .

Aunque para problemas sencillos C también puede ser una opción, (pero ya que C++ es casi casi un superset de C podrías entonces igual de fácil hacerlo en C++).

2.2.2. vs Java

Con toda honestidad hay un porcentaje de la comunidad de programación competitiva que usan Java, así que si que es una opción viable, sobretodo por su gran libreria estándar y también porque en C++ no hay algo parecido a `BigInteger` y `BigDecimal` y suelen ser muchos los problemas que lo requieran, así que si bien C++ podría ser tu lenguaje por defecto es importante que también conozcas lo básico de Java (O Kotlin si quieres ser feliz).

2.2.3. vs Python

Python es un gran lenguaje pero tiene todas las de perder en programción competitiva pues a ser interpretado y debilmente tipado, sus programas acaban siendo muy lentos incluso usando el algoritmo correcto, eso si, hay varias aplicaciones útiles de Python, como que todos los enteros tienen infinita precisión por defecto (aka `BigInteger` como en Java).

Así que tampoco es una mala idea aprenderlo por si se necesita un día, pero definitivamente no es la mejor idea para ser tu lenguaje por defecto en programación competitiva.

2.3. Una pincelada de C++ moderno

Incluso en términos solo de sintaxis te perdonaría si pensaras que C++ es un lenguaje terminado, algo que se hizo en los 90's y que seguimos escribiendo igual al día de hoy.

Y no es así, C++ 11 / 14 / 17 / 20 son cosas muy diferentes, igual de flexibles y de rápidas que el clásico C++ 98 pero mucho mas seguro y limpio de escribir.

Mira un ejemplo.

```
//Old C++
circle* p = new circle(42);
vector<shape*> v = load_shapes();

for (vector<shape*>:::iterator i = v.begin(): i != v.end(); ++i) {
    if (*i && **i == *p)
        cout << **i << "is a match" << '\n';
}

// ...later, possible elsewhere

for (vector<shape*>:::iterator i = v.begin(): i != v.end(); ++i) {
    delete *i;
}

delete p;
```

Mientras que ahora podrías hacer algo como:

```
//New badass C++
auto p = make_shared<circle> (42);
auto v = vector<shape*> {load_shapes()};

for (auto& s : v) {
    if (s and *s == *p)
        cout << *s << "is a match" << '\n';
}
```

Veamos otro ejemplo, imagina que alguien te da una secuencia de puntos (flotantes por ejemplo) y te pide calcular su media.

Veamos como sería hacerlo en Python así de volada:

```
def mean(seq):
    n = 0.0
    for x in seq:
        n += x
    return n / len(seq)
```

Y en C++ mira como sería:

```
auto mean(const Sequence& seq) {
    auto n {0.0};
    for (auto x : seq)
        n += x;
    return n / seq.size();
}
```

Que bonito, ¿no? [2]

Capítulo 3

Instalando y corriendo programas de C++

3.1. Instalar C++

3.2. Corriendo C++

3.3. Jueces en línea

Capítulo 4

Bases del Lenguaje Moderno

Recuerda que esta sección del texto NO es una introducción a la programación para alguien que nunca ha programado nada en su vida, sino solo para alguien que no sabe `C++` o que no sabe todo de `C++` .

Si tu aún no sabes nada sobre como programar entonces recomiendo que busques un documento, tutorial, libro, etc... diseñado para empezar a programar, porque en este texto voy a dar varias cosas por sentado que deberían ser muy obvias para alguien que ya haya aprendido a programar, en cualquier lenguaje.

Unas recomendaciones serían:

- **Curso de programación básico de Platzi**
- **Every Programming Language in 15 Minutes - Brian Will**

No te preocupes, te espero `<3` .

Además si ya sabes `C++` o no te interesa aprender toda la sintaxis e ir directo a cosas algo más relacionadas con la programación competitiva entonces puedes saltar hasta el siguiente capítulo dando click aquí.

4.1. Variables

En C++ (como en casi cualquier otro lenguaje) la idea obvia con la que podemos empezar es las variables, después de todo la programación siempre se trata sobre información (data) y podemos ver a una variable como una unidad de información.

En C++ una variable es un fragmento de memoria que almacena algun valor, podemos tener variables que almacen lo que nosotros entenderemos, como números enteros o que almacenen cadenas o que almacenen una secuencia de racionales, etc...

Como sabes en programación, tú eres el que elige el nombre de las variables y aunque puede ser lo que quieras, como `x, y, someNumber, patata` intenta elegir nombres que expresen lo que esta variable guardará como por ejemplo `numberOfDays, sizeOfSquare, words, isTurnOfAlice` .

Bueno, C++ es un lenguaje de tipado estatico (static typing), es decir que todo momento el compilador, para poder transformar tu programa a algo que una computadora pueda entender, tiene que saber que tipo de dato almacena esa variable.

4.1.1. Declaraciones e Inicialización

El primer paso para poder usar una variable será el de declararla, es decir aquí entre nos es decirle al compilador que le de a un fragmento de memoria un nombre y que usaremos ese nombre (o identificador) para referirnos a ella después.

La sintaxis es sencilla:

```
datatype variableName;

//Examples
int numberOfDays;
float pi;
bool isClosed;
```

Ahora, algo importante en C++ es que declarar una variable no la inicializa, es decir, que al decir `int numberOfDays` nunca le estoy dando un valor a esa variable, por lo que ahora mismo la información que este guardada ahi es un misterio, es como si comprarás una bodega; solo por decir que es tuya no quiere decir que ahora este vacía.

Por eso es tan poco común ver en la programación competitiva (y en la programación en general) declaraciones SIN inicializaciones, pues generan una gran cantidad de bugs que son difíciles de encontrar pues muchas veces por simple suerte la variable si se inicializa a un valor que tenga sentido, pero al ser este proceso aleatorio no es una gran idea depender de eso.

Además es una buena costumbre, no declarar variables hasta que tenga un valor coherente para las mismas.

Muy unido a esto decimos que estamos inicializando una variable cuando le damos un valor por primera vez a este fragmento de la memoria.

Puedes declarar variables en C++ de dos maneras:

Se directo con el tipo de dato

Algo como:

```
int someNumber = 20;          //Good: declaration + initialization
string someText = "Hi baby";  //Good: declaration + initialization
double myLoveForYou;          //Bad: just declaration
```

Es decir, la sintaxis es:

- Primero el tipo de dato, después un espacio.
- Después el nombre que le quieres dar a la variable.
- Si quieres un valor inicial.

auto: Se sútil

Si es que es obvio que tipo de dato debería ser entonces puedes usar `auto` que lo que nos dice es: *compilador, yo se que eres un inteligente, anda, tu solito sabes de que tipo de dato es esta variable; ¿para que te lo repito yo?*

Y la sintaxis es bastante similar:

```
auto someNumber = 20;         //someNumber is int
auto someText = "Hi baby";    //someText is const char* (this is sad)
auto myLoveForYou;            //This will not compile :v
```

Nota que para que puedas usar `auto` el compilador tiene que saber que tipo de dato va a guardar esa variable así que si no inicializas la variable el compilador se va a enojar contigo.

De igual manera creo que te habras dado cuenta que podemos inicializar de dos maneras generalmente la primera es muy obvia y en casi todos los lenguajes existe, se llama **una asignación**, es decir, darle un valor a esa variable y se usa casi siempre en todos los Lenguaje el símbolo `=` o a veces incluso `<-` .

Total, lo que pasa en `C++` es que además de esa forma de inicializar variables tenemos una forma que si tiene un nombre especial.

Uniform (brace) Inicialization: Inicialización uniforme

Y lo que nos da esto es una misma sintaxis, quiza esto sea un tema algo complejo para unos y no te preocupes si no entiendes esto por completo por ahora.

Total, lo que pasa es que en `C++` moderno existe la sintaxis `type wea {something}` donde lo que hacemos es poner entre estas cosas `{ }` el valor con el que queremos inicializar nuestra variable y dependiendo de que sea nuestra variable puedes simplemente asignarla o llamar al constructor con estos parámetro.

Total, es solo una forma mas bonita de hacer las cosas.

```
auto someNumber {20};
string someText {"Hi baby"};
// this call a someClass constructor
someClass object {"some parameter", someNumber};
vector<int> numbers {1, 2, 3, 4, 5, 6};
```

Además ayuda en varias cosas:

- Supón que quieres inicializar por defecto un vector, entonces puedes hacer como:

  ```
  vector<int> days ();
  ```

  Expecto que esto es la declaración de una función llamada `days` que no recibe nada y regresa un vector.

  En la antigüedad podríamos hacer algo como:

  ```
  vector<int> days = vector<int>();
  ```

  Pero ahora algo mucho mas sencillo gracias a uniform inicialization:

  ```
  vector<int> days {};
  ```

4.2. Tipos de datos numéricos

4.2.1. Integers / Integrals: Enteros

C++ (y C) maneja varios tamaños estándares de enteros, el más clásico es `int` , pero no es el único, los demás solo cambian en tamaño (y por los tanto los números que podemos almacenar) y estos son, de menor a mayor: `char, short, int, long` y `long long` .

Así que con esto conocido, veamos las características más detalladamente de cada tipo: Pero si quieres un resumen, C++ garantiza que:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <=
    sizeof(long long);
```

char

A ver, técnicamente este tipo de dato como su nombre lo dice esta diseñado para almacenar un carácter, lo que pasa es que en computación (no nos compliquemos la vida con UFT ahora) un carácter como `'a'` ó `'p'` se representa internamente usando el código ASCII (deberías buscar más de esto cuando tengas tiempo).

Total, que en resumen este tipo de dato nació para almacenar un carácter de ASCII, por lo tanto es un tipo de datos numérico que va de 0 a 255.

No te preocupes si no entiendes las primeras líneas, pero si lo haces, muy bien por ti.

```
auto character {'a'};                    // character is a char!
char number {23};                        // number is a char!

// Examples: Things that are true
('a' == 97) and ('z' == 122);            // ASCII is just numbers
('A' == 65) and ('Z' == 90);             // ASCII is just numbers
('A' + 1 == 'B') and ('Z' - 1 == 'Y');   // You can do arithmetic

auto size {"char is 1 byte"};
char minValue {-128};
char maxValue {127};
```

También es importante recalcar que podemos hacer aritmética con variables de este tipo de dato como lo muestro en los ejemplos.

int

Es el tipo de dato numérico estándar en C++ , así que si declaras un número con `auto` entonces lo más probable es que sea `int` .

Ahora, pasa algo raro con este tipo de dato, que dependiendo de la máquina en la que compiles entonces puede tener 2 o 4 bytes de tamaño (usa el que sea más eficiente para el sistema), aun así, casi nunca compilaras en algún sistema que te diga que un `int` es de 2 bytes, así que para este texto usaremos que `int` es de 4 bytes.

```
auto number {23};                              // number is an int!

auto size {"int is 4 bytes"};
int minValue {-2,147,483,648};
int maxValue {2,147,483,647};
```

long long

Este no es un tipo de dato, per se, sino que es un modificador que le puedes aplicar a `int` y con esto lo que logras es crear un tipo de dato que tiene al menos 8 bytes.

Por eso es que ignore a `long int / long` pues solo nos garantiza que tendra al menos 4 bytes por lo que es mas o menos inútil, lo mas seguro es declarar algo como `long long` si te interesa que tenga el máximo tamaño posible.

```
int normalVariable {};                         // It takes 4 bytes
long long int normalVariable {};               // It takes 8 bytes
auto number {1,000,000,000,000,000,000}        // number is long long

auto size {"long long int is 8 bytes"};
long long minValue {-9,223,372,036,854,775,808};
long long maxValue {9,223,372,036,854,775,807};
```

unsigned

Lo que hace este modificador (exacto, esto tampoco es un tipo de dato) es eliminar el signo de los tipos numéricos, es decir el entero más bajo que vas a poder guardar va a ser el 0, pero con ello vas a lograr duplicar el máximo entero que puedes almacenar y no aumentas para nada el espacio necesario :o

```
char maxValueChar {127};
unsigned char maxValueIntUnsigned {255};

int maxValueInt {2,147,483,647};
unsigned int maxValueIntUnsigned {4,294,967,295};

long long maxValueLL {9,223,372,036,854,775,807};
unsigned long long maxValueLLUnsigned {18,446,744,073,709,551,615};
```

short

Hace lo inverso que `long` , en vez que duplicar el tamaño lo parte a la mitad, y ya, solo eso :v

```
auto size {"short is 2 bytes"};
short int minValue {-32,768};
short int maxValue {32,767};
```

std::size_t

Este es especial y muchas veces lo usaré como estándar de tipo numérico y es que usamos muchas veces los enteros como índice de un contenedor.

Bien pues `size_t` es un tipo de dato especial que nos da `C++` que nos asegura que será tan grande como necesitemos para usarlo como índice de cualquier contenedor.

Nota que como lo usamos para índice, este tipo de dato no tiene signo y generalmente tiene el mismo tamaño de que un `long long` .

Además si te gusta los detalles visita esto: aquí.

Por ejemplo, `std::vector, std::string, std::array` y más lo usan como índice.

```
std::vector<int> someIntegers {1, 2, 3, 4, 20, 5};
std::size_t numberOfElements {someIntegers.size()};
```

Fixed width (int32_t, uint64_t, ...)

Si prefieres estar seguro del tamaño de tus enteros entonces puedes usar `#include <cstdint>` que no incluye otros tipos de datos diferentes sino solo nos da alias (typedef / using), es decir otros nombres para los tipos de datos que ya conoces:

```
#include <cstdint>

int8_t likeChar {};
int16_t likeShort {};
int32_t likeInt {};
int64_t likeLong {};

// And the unsigned versions:
uint8_t likeChar {};
uint16_t likeShort {};
uint32_t likeInt {};
uint64_t likeLong {};
```

4.2.2. Floating Points: Puntos Flotantes

A primera vista, los números de punto flotante parecen simples. Son solo enteros con puntos decimales, ¿verdad? ¿Por qué no los usamos todo el tiempo, ya que pueden almacenar una mayor variedad de números?

La respuesta es sencilla, no son precisos.

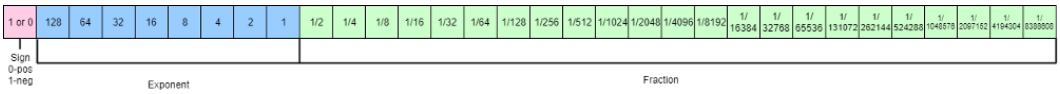


Figura 4.1: Representación del `float` en C++

Y ya, solo por eso, intenta poner en Python o en C++ si quieres esto `(0.1 + 0.2) == 0.3` y verás que es falso, la razón es que los números de punto flotante tienen una cantidad límitada de dígitos de presición.

Así que esta bien usarlos y todo, pero porfavor, hazlo con cuidado.

Además así como `long long` era el doble que `int` , así `double` tiene el doble de presición (de ahí el nombre :v) que el clásico tipo de dato de punto flotante en C++ , `float` .

```
float almostPI {3.1};
double almostAlmostPI {3.14156};
```

4.2.3. Complex: Números Complejos

Esta sección igual será corta, lo único que quiero decirte por si un día lo ocupas es que C++ tiene una clase que nos permite representar a los complejos.

Nunca lo he ocupado de manera personal y no creo que sea el momento, en un texto introductorio de C++ , pero si un día lo necesitas, recuerda que C++ ya lo tiene integrado.

4.2.4. Utilidades

Puedes obtener el máximo y el mínimo de cualquier tipo de dato entero así:

```
#include <iostream>     // std::cout
#include <limits>       // std::numeric_limits

int main () {
    const auto minVal = std::numeric_limits<int>::min();
    const auto maxVal = std::numeric_limits<int>::max();

    std::cout << "Minimum value for int: " << minVal << '\n';
    std::cout << "Maximum value for int: " << maxVal << '\n';

    return 0;
}
```

4.3. Los otros tipos de datos fundamentales

4.3.1. void

Este es bastante sencillo, (si no cuentas `*void`), y simplemente indica un tipo de dato vacío, es usado para mostrar que una función no regresa nada.

Decimos que `void` tiene un conjunto vacío de elementos, es decir no puedes hacer variables (o instanciar) cosas de tipo `void` por lo que muchas veces se le llama un tipo de dato incompleto.

```
void someVariable;  // Illegal
```

4.3.2. bool

Solo puede tener dos posibles valores, verdadero o falso, se llama así por el álgebra booleana, que justamente solo contaba con dos valores.

```
bool isSunny {true};
bool iAmHappy {false};
```

Cambiar el valor de un bool

Como un `bool` solo puede tener un valor muchas veces queremos andar volteando los valores de un valor booleano, es decir, primero eres `true` , luego `false` , luego `true` , luego `false` , luego `true` , etc...

Para hacerlo lo que hacemos sera negar el antiguo valor que tenian y guardar ese nuevo valor, esto se hara con algo como:

```
bool isEven {true};
int n {};
while (true) {
    cout << n << " " << isEven << '\n';

    isEven = not isEven;        // The important line
    ++n;
}
```

4.4. Operadores

4.4.1. Operadores Aritméticos

Sección corta, si sabes programar los has usado:

- `+` : Suma
- `-` : Resta
- `*` : Multiplicación

Además en C++ tenemos algunas ayudas para escribir menos:

- `a++` es lo mismo que `a = a + 1` , ademas decimos que es una expresión regresa a `a` antes de hacer la suma.

```
int a {1};
int b {a++};     // b is 1
```

- `a--` es lo mismo que `a = a - 1` y con las mismas reglas que `a++`
- `++a` es lo mismo que `a = a + 1` , ademas decimos que es una expresión regresa a `a` después de hacer la suma.

```
int a {1};
int b {++a};     // b is 2
```

- `a += b` es lo mismo que `a = a + b`
- `a -= b` es lo mismo que `a = a - b`
- `a *= b` es lo mismo que `a = a * b`
- ...

Ahora los interesantes son los siguientes:

4.4.2. División entera

Una cosa que a mucha gente le cuesta es la división y no porque sea una operación difícil sino porque en C++ (y en muchos otros lenguajes de programación) dependiendo del tipo de dato de las variables a la que se lo apliquemos se comportara de manera o de otra.

Mira:

```
std::cout << 5 / 2 << std::endl;    // prints 2     :o
std::cout << 5.0 / 2 << std::endl;  // prints 2.5   :o
```

Las reglas son:

- Si es que al menos uno de ellos es un punto flotante entonces hara lo que esperas por ejemplo `1.0 / 2.0 = 0.5`
- Si es que ambos son "integrals" (es decir enteros), entonces lo que haraá seraá hacer un redondeo hacia abajo, conocido también como piso, es decir va a redondeo al entero proximo mas pequeño.

Hay que tener cuidado con eso.

4.4.3. Lo que hay que saber del módulo: %

El módulo es pocas palabras es el residuo de la división.

Es decir `a% b` regresa el residuo de la división $\frac{a}{b}$ si la has visto entonces verás que es la misma que en todos los lenguajes sino entonces mas vale que practiques con unos ejemplos, es todo cuestión de practica.

La respuesta para los puristas

Si eres matemático y quieres la definición formal podríamos pensar en:

$$a\,\%b = a - floor(a/b) \ * \ b$$

Otra forma de verlos es que nos regresa un número `k` tal que $k \equiv a \mod b$

Ejemplos

- $7\,\%5 = 2$

- $5\,\%7 = 5$
- $3\,\%7 = 3$
- $2\,\%7 = 2$
- $1\,\%7 = 1$

La forma mas común para lo que lo usamos es para saber si un número es par o impar, donde si n es par entonces `n% 2 == 0` y si es impar entonces `n% 2 == 1` .

En general, si quieres saber si un número n es multiplo de otro k entonces hay que checar que `n% k == 0` .

4.4.4. Operadores Relacionales

Estos sirven para comparar los elementos, también son practicamente iguales a como son en todos los lenguajes:

- Igualdad Lógica:

  Se escribe así y solo el valor de verdad de ambos valores es igual.

```
bool IAmHappy = IHaveLove == true
```

- Desigualdad Lógica:

  Se escribe así y solo el valor de verdad de ambos valores es diferente.

```
bool IAmHappy = IHaveLove != true
bool IAmHappy = IHaveLove not_eq true
```

- Mayor y menor:

```
int debt{20}, money{50};
bool IAmHappy = debt < money;
bool IAmHappy = money > debt;
```

- Mayor / menor o igual:

```
int debt{20}, money{50};
bool IAmHappy = debt <= money;
bool IAmHappy = money >= debt;
```

4.4.5. Operadores Lógicos

En `C++` tenemos varios operadores lógicos que tomaran como argumentos dos `bool` o cosas que se puedan convertir a bool y con ellas nos darán otro valor booleano.

- AND Lógico:

  Se escribe así y solo será verdad si es que ambos parámetros son `true` .

```
bool IAmHappy = IHaveAJob && IHaveAFreeTime
bool IAmHappy = IHaveAJob and IHaveAFreeTime
```

- OR Lógico:

  Se escribe así y solo será verdad si es que al menos uno de los parámetros son `true` .

```
bool IAmOk = IHaveAJob || IHaveAFreeTime
bool IAmOk = IHaveAJob or IHaveAFreeTime
```

- NOT Lógico:

  Se escribe así y solo invierte el valor de verdad de lo que le enviemos.

```
bool IAmHappy = !IAmSad
bool IAmHappy = not IAmSad
```

4.4.6. Operadores Lógicos bit a bit

En C++ tenemos varios operadores lógicos bit a bit que tomaran como argumentos dos números y nos regresan otro numero.

Ahora, los demás operadores son tan obvios que practicamente no tengo nada que decir, pero para estos si que tengo mucho que decir, y antes de que diga mas hay que recordar el binario, y como funciona, asi que si algo te suena raro recuerda checar si estas entendiendo bien el binario, ahora si, veamoslos con cuidado:

- AND Lógico bit a bit:

  Se escribe así, y lo que hace es aplicar AND bit a bit de nuestros números, es decir un bit es 1 si y solo si es que ambos son 1.

```
int number1 {0b11100011};   // C++14
int number2 {0b01010101};   // C++14

int numberAND = number1 & number2;
int numberAND = number1 bitand number2;
```

  Donde tendremos que `numberAND = 0b01000001` . Mira porque:

```
  11100011    -> number1
& 01010101    -> number2
---------
  01000001
```

- OR Lógico bit a bit:

  Explicado el AND logico, este es basicamente el mismo, pero que ahora siguiendo el OR, es decir, solo si alguno de los dos es uno entonces el valor de ese bit será uno.

```
int number1 {0b11100011};   // C++14
int number2 {0b01010101};   // C++14

int numberOR = number1 | number2;
int numberOR = number1 bitor number2;
```

  Donde tendremos que `numberOR = 0b11110111` . Mira porque:

```
  11100011    -> number1
& 01010101    -> number2
---------
  11110111
```

- XOR Lógico bit a bit:

  Lo mismo pero siguiendo ahora el XOR bit a bit:

```
int number1 {0b11100011};   // C++14
int number2 {0b01010101};   // C++14

int numberXOR = number1 ^ number2;
int numberXOR = number1 xor number2;
```

  Donde tendremos que `numberXOR = 0b10110110` . Mira porque:

```
  11100011    -> number1
& 01010101    -> number2
---------
  10110110
```

- Negación Lógica bit a bit:

  Lo que hace es negar cada bit:

```
int number {0b11100011};   // C++14
int numberNeg = ~number;
int numberNeg = compl number;
```

  Donde tendremos que `numberNeg = 0b00011100` . Mira porque:

```
11100011    -> number
---------
00011100
```

- Shifts bit a bit:

  Toma dos números y lo que hace es desplazar el primer número la cantidad que dice el segundo número de veces y llena lo que va moviendo con 0's, veamos las dos versiones que tenemos:

  - Left shift

```
int number {0b00000110};
int number1 = number << 1;      // 0b00001100
int number2 = number << 3;      // 0b00110000
```

  - Right shift

```
int number {0b00000110};
int number1 = number >> 1;      // 0b00000011
int number2 = number >> 2;      // 0b00000001
int number3 = number >> 3;      // 0b00000000
```

4.5. Sentencias de Control

4.5.1. Branching: Condicionales

Sin una declaración de un condicional como la instrucción `if` , los programas se ejecutarian siempre de la misma manera.

Los condicionales permiten que se cambie el flujo del programa y con ello códigos más interesantes.

```
//Simpler form
if (condition) {
    ...
}

//Simple form
if (condition) {
    ...
}
else {
    ...
}

//Complete form
if (condition1) {
    ...
}
else if (condition2) {
    ...
}
else if (condition3) {
    ...
}
else {
    ...
}
```

Condiciones

En C++ lo que puede ir dentro del `if (condition)` pueden ser dos cosas:

- Una expresión que se pueda transformar a un `bool` y eso es la cosa mas común.

  Por ejemplo:

```
bool isSunny {true};
if (isSunny == true) {
    cout << "Hey, time to go to outside" << '\n';
}

if (isSunny) {
    cout << "Hey, it is the same but shorter syntax" << '\n';
}

double pi {3.1416}, e {2.71828};
if (pi > e) {
    cout << "Math still make sense" << '\n';
}
```

  Tambien podemos poner dentro del if una variable lo que hara que C++ transforme el valor de dicha variable en un booleano, las reglas que sigue son bastante sencillas:

```
if (n) {
    ...
}
```

  - Para números: Si n es cero entonces el if evaluara a falso, para cualquier otro valor será verdadero.
  - Para strings: Si n esa vacío entonces sera falso, cualquier otra otro valor será verdadero.
  - Para punteros: Si n es NULL / nullptr entonces sera falso, cualquier otra otro valor será verdadero.

- Se puede hacer `if (a = b)` en cuyo caso lo que comparará el `if` es el resultado de la asignación, es decir checa si `b` es "verdadero".

Operador Ternario

En C++ tenemos el operador ternario, es bastante común porque nos deja expresar la misma idea que un `if` pero mucho mas corto, además es una expresión, es decir regresa un valor, esa es la más grande referencia.

Y por lo tanto es muy útil cuando lo único que hacemos es darle un valor a una variable o algo relativamente sencillo.

```
// Using if else
auto programmerFeelings = "";
if (language == "php") {
    programmerFeelings = "sad";
}
else {
    programmerFeelings = "happy";
}

// The same thing using ternary operator
auto programmerFeelings = (language == "php") ? "sad" : "happy";
```

4.6. Ciclos

En C++ tenemos distintos tipos de ciclos, y si funcionan exactamente igual que en todos los demás lenguajes, tenemos:

4.6.1. While

```
while (condition) {
    // statements
}
```

Donde condition es exactamente igual que lo que estaría dentro del if.

Y de hecho funciona basicamente igual que un if, la única diferencia es que una vez que se haya ejecutado el cuerpo entonces volveremos a ver la condición, de ser verdadera el cuerpo se volverá a ejecutar.

Do while

Existe también una variante que se llamada `do while` y que es bastante menos conocido que el clásico `while`, así que creo que vale la pena hablar un poco de el.

```
do {
    // statements
}
while (condition);
```

En este ciclo primero ejecutamos las instrucciones que esten dentro del bloque y luego es que checamos la condición y si es verdadera entonces volvemos ejecutar las instrucciones.

4.6.2. For

Igual que en cualquier otro lenguaje:

```
for (init expr; condition expr; step expr) {
    // statements
}
```

Que como sabes si sabes programar es exactamente igual que:

```
init expr;
while (condition expr) {
    // statements
    step expr;
}
```

Ahora, hablemos del nuevo pequeño ciclo en el lenguaje:

4.6.3. Range for: `for (auto x : v)`

En C++ tenemos otra forma más moderna y muchos diran que mucho mas fácil de evitar bugs raros si lo único que haremos será movernos un elemento a la vez a lo largo de un contenedor, y si en efecto vamos a visitar cada elemento desde el inicio hasta el final del contendedor (no te preocupes, pronto hablaremos sobre contenedores) entonces puedes hacer un clásico:

```
for (type element : container) {
    // statements
}
```

Por ejemplo:

```
vector<int> someNumbers {1, 2, 3, 4};

for (int i : someNumbers) {
    cout << i << '\n';
}
```

Y si lo que quieres es iterar sobre elementos que pueden ser caros de copiar entonces podemos también usar una referencia (si no sabes que son, no te preocupes, pronto hablaremos de eso) de este y tener algo como:

```
vector<string> countryCodes {"USA", "MEX", "JAN", "UK", "CHL", "SA"};

for (string& countryCode : countryCodes) {
    cout << countryCode << '\n';
}
```

Tambien sirve si es que quieres alterar los elementos del contenedor:

```
vector<int> numbers {1, 2, 3, 4};

for (int& number : numbers) {
    number *= 3;
}

// Now numbers is [3, 6, 9, 12]
```

Y si, hablaremos sobre referencias y `auto&&` más a detalle pronto.

Además puedes usar `break, continue` como en cualquier `for` normal

```
vector<int> numbers {1, 2, 3, 4};

for (int& number : numbers) {
    if (number == 3) break;
}

for (int& number : numbers) {
    if (number != 3) continue;
    else {
        ...
    }
}
```

4.7. References and value types

4.7.1. Value types: Variables de valor

En C++ como llevamos diciendo todo el libro las variables por defecto son de tipo valor, es decir que cada variable almanena de verdad sus valores.

```
int someNumber {3};
```

Pero no solo podemos expresar esa idea, sino que también podemos expresar la idea de una "referencia a una variable".

4.7.2. References: Referencias

Las referencias solo son otro nombre para una variable, es como un alias.

Se usan cuando uno quiere estar pasando el valor de una variable sin tener que copiar la misma variable, recuerda: No es gran cosa copiar una pequeña variable, pero copiar contenedores (como vectores, arreglos, strings, etc...) pueden ralentizar mucho un programa.

Además si sabes de C y sus punteros, entonces te puedo decir que las referencias cumplen una función parecida, pero también hacen el código más sencillo de leer al evitar cosas como los operadores de indirección: `*` y el operador flecha `->`.

Una referencia se crea así:

```
int number {};
int& referenceToNumber {number};
```

Ahora, una variable de tipo referencia esta unida desde el momento en el que se crea a otra. No puedes hacer que una variable de tipo referencia ahora haga referencia a otra variable después de ser creada, además una referencia siempre tiene que referir a una variable, no puede referir a la nada.

Problemas que solucionan

Las referencias vienen a solucionar dos problemas bastante clasicos en programación, sobretodo en C++ :

- **Hacer que las funciones afecten los argumentos que les pasas y no a copias**

  Veamos un ejemplo, supongamos esta función:

```
void add_3_to_this_number(int number) {
    number += 3;
}
```

  Entonces si hacemos algo como esto, ¿Qué esperas que pase?

```
int the_number {0};
cout << the_number << '\n';        // prints 0
add_3_to_this_number(the_number);
cout << the_number << '\n';        // prints 0 :c
```

  Si conoces como funcionan las funciones en C++ entonces es claro que `the_number` sigue valiendo 0, porque cuando usas una función, estas de verdad creando una copia y es a esa copia a la que estas añadiendole 3.

  Si quieres crear una función que de verdad tome a un entero y le sume 3 a ESE entero, entonces podemos hacer que reciba el parámetro por referencia:

```
void add_3_to_this_number(int& number) {
    number += 3;
}

// later ...

int the_number {0};
cout << the_number << '\n';        // prints 0
add_3_to_this_number(the_number);
cout << the_number << '\n';        // prints 3 :)
```

- **Evitar generar copias a lo estúpido**:

  Encontre este texto en internet y no pude evitar compartirlo:

  " `You live in a house on a street in a town somewhere. I want to visit you to give you something.`

  `I have two choices - you can tell me your address, and I'll come to you, or you can bring your house on the back of a huge truck to me. One of these is obviously ridiculous.` "

  O en español:

  " `Tu vives es una casa, en una calle cualquiera en algún pueblo. Supón que te quiero visitar para darte algo.`

  `Tienes dos opciones - me puedes decir tu dirección y yo ire contigo o puedes traer tu casa en un camión enorme hasta mi. Una de ellas es obviamente ridícula.` " [3]

  Veamos otro ejemplo, supongamos esta función:

```
int sumTheFirstTwoElements(vector<int> numbers) {
    int result = numbers[0] + numbers[1];
    return result;
}
```

  Ahora, a diferencia del ejemplo de arriba no importa si es que `numbers` es el vector en si o una copia, pero si es que estamos generando una copia entonces estamos haciendo nuestro código mucho mas lento sin razón.

  Ahora, podemos solucionar esto de manera muy sencilla como:

```
int sumTheFirstTwoElements(vector<int>& numbers) {
    int result = numbers[0] + numbers[1];
    return result;
}
```

  Ahora, si tienes miedo de pasar cosas por referencias por el miedo a que la función modifique tu argumentos entonces podemos hacer algo tan sencillo como:

```
int sumTheFirstTwoElements(const vector<int>& numbers) {
    int result = numbers[0] + numbers[1];
    return result;
}
```

References vs Pointer: Referencias vs Punteros

- **Te evita problemas con `nullptr` / `NULL` :**

  Ve la siguiente función:

```
int foo(bar* b1, bar* b2) {
    return b1->calculate(b1->get());
}
```

  Ahora, nada evita que pueda llamar a la función con algo como:

```
int result = foo(NULL, NULL);           // If you like C
int result {foo(nullptr, nullptr)};     // If you like C++
```

  Y el compilador no te dira nada, pero si hago algo como lo siguiente puedo solucionar el problema:

```
int foo2(const bar& b1, const bar& b2) {
    return b1.calculate(b1.get());
}
```

  Entonces el compilador se va a enojar con nosotros si es que escribimos algo como:

```
int result = foo2(NULL, NULL);          // Compile Error
int result {foo2(nullptr, nullptr)};    // Compile Error
```

- **El código con referencias es mucho mas sencillo que con los punteros**:

  Por ejemplo, estos dos códigos son mas o menos equivalentes, pero a que no adivinas cual es mas claro:

```
void swap(int &a, int &b) {
    int tmp {a};
    a = b;
    b = tmp;
}

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- **No puedes hacer aritmetica de punteros con las referencias**:

  Es decir, no puedes hacer cosas como `*(a+1)`

- **No puedes re-asignar una referencia**.

  Una gran diferencia es que una referencia no se puede re asignar, por eso es que los punteros son una idea perfecta para implementar estructuras de datos como listas enlazadas, árboles, etc ...

- **No puedes tener referencias a referencias**.

  Puedes tener punteros a punteros pero las referencias solo tienen un nivel de indirección, por lo tanto es un error crear una referencia de referencias.

4.8. Funciones

Una función es un conjunto de instrucciones (statements) que podemos llamar desde cualquier parte de nuestro programa y que opcionalmente puede tener algunos parámetros y de igual manera podemos regresar un valor desde nuestras funciones.

Una función en C++ son practicamente iguales que en los demás lenguajes, veamos su sintaxis.

```
returnType nameOfFunction(argType1 name1, argType1 name2, ...) {
   body
}
```

Por ejemplo:

```
int doubleIt(int number) {
    return number * 2;
}
```

Es decir, primero va el tipo de dato que regresará nuestra función, después el nombre que le daremos y finalmente entre paréntesis una lista de parámetros con el nombre que le daremos y el tipo de dato de dichos parámetros.

Además desde C++ 11 hay otra forma de declarar funciones:

```
auto nameOfFunction(argType1 name1, ...) -> returnType {
   body
}
```

Ambas son exactamente igual.

```
auto doubleIt(int number) -> int {
    return number * 2;
}
```

Además con esta nueva sintaxis incluso podemos ignorar tipo de dato de regreso si es que el compilador puede inferirlo.

Ambas son exactamente igual.

```
auto doubleIt(int number) {     //Since C++14
    return number * 2;
}
```

Ademas hace que todo vaya mas en sintonía con las lambdas.

4.9. Tips

- `inline`

  Si tu función es algo que podrías reemplazar en donde la llamaste, algo como:

```
// Before
#include <iostream>
auto cube(int s) { return s * s * s; }

int main() {
    std::cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

Entonces puedes usar `inline` que lo que hace es darle una pista al compilador de que ni siquiera haga todo el proceso de llamar a la función sino simplemente copie y pegue su contenido.

```
// After
#include <iostream>
inline auto cube(int s) { return s * s * s; }

int main() {
    std::cout << "The cube of 3 is: " << cube(3) << "\n";
    // Same as:
    std::cout << "The cube of 3 is: " << 3 * 3 * 3 << "\n";

    return 0;
}
```

4.9.1. Lambdas

Una lambda es un concepto bastante conocido en otros lenguajes, podemos definirlo como la una función que puede ser tratada como cualquier otro objecto en el lenguaje.

Las lambdas son un nombre bonito para funciones anónimas, en esencia son una forma fácil de escribir funciones en el lugar lógico que deberían estar.

Estan diseñadas para ser usadas como callbacks o para poder pasar una función como parámetro a otra función.

Su sintaxis es algo como:

```
[capture list] (parameter list) -> returnType { body };
```

Donde:

- `capture list` es una lista de valores que deben ser visibles dentro de la lambda y si deberían tomarlos por valor o por referencia.

  Estos seran copiados o se creara una refencia a ellos en el momento en que se declara.

  Puedes decirle al compilador que capture lo que sea necesario por:

  - valor con: `[=](...) { body }`
  - referencia con: `[&](...) { body }`
  - mezcla por ejemplo: `[&, i](...) { body }` para decir todo por referencia, excepto i que es por valor.

- `parameter list` funciona igual que una lista de parámetro por una referencia normal lo que estamos haciendo es una promesa, prometemos que quien haya llamado a de una función.

Por ejemplo para ordenar de reversa un vector podemos hacer algo como:

```
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
```

Veamos otro ejemplo:

```
int var = 1;
auto someFunction = [var] () {
    cout << "Hello from lambda, value is " << var << '\n';
};

someFunction(); // output: 1
var = 4;

someFunction(); // output: 1
```

Y un último:

```
int var = 1;
auto someFunction = [&var] () {
    cout << "Hello from lambda, value is " << var << '\n';
};

someFunction(); // output: 1
var = 4;

someFunction(); // output: 4
```

También es importante saber que las lambdas no son de tipo `std::function` sino que pueden ser asignadas a una, pero en el fondo son syntactic sugar para objectos, functors.

Además su rendimiento es muy bueno, porque al ser objectos el compilador puede optimizar muchas veces mas el código que si hubieramos usado una función de toda la vida.

Lambdas como functors / classes

Puedes pensar en las lambdas como en clases, y no te preocupes si no entiendes esta sección ahora, puedes saltarla si quieres.

La lista de valores capturados son como los elementos de una clase es decir si tenemos algo como esto:

```
[&total, offset](int element) {
    total += element + offset;
};
```

Podriamos verlo como si estuvieramos haciendo algo como:

```
class lambda {
    private:
        int& total;
        int offset;
    public:
        lambda(int& _total, int _offset) :
            total{_total}, offset{_offset} {}

        void operator() (int element) const {
            total += element + offset;
        }
};
```

[6]

4.10. Templates

Uno de los temas más importante en este sección es hablar sobre esto. Primero te mostraré un problema:

Supongamos que quieres hacer una función llamada ReLU, algo como:

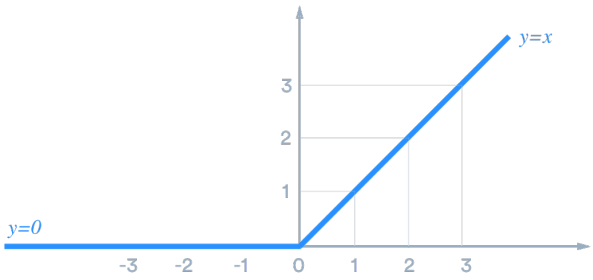


Figura 4.2: ReLU: Rectified Linear Unit - una función usada en el machine learning

Podiamos implementarla así:

```
double ReLU(double input) {
    return input > 0? input : 0;
}
```

Ahora, supongamos que tenemos que soportar que esta función acepte `int, float, etc...` . ¿Qué hacemos? ¿hacer 3 funciones iguales?

Si estas con la mentalidad de C entonces puedes pensar:

```
// FUCK*NG MACROS!
#define ReLU(a)((a) > 0? (a) : 0)
```

Pero... primero que nada, esto se ve horrible, y segundo, no estas creando una función sino esta haciendo un reemplazo de texto, por lo que puede que pasen cosas medio raras:

```
auto a = 3.0;
auto result = ReLU(a);  // Works result = 3
//same as:
auto result = (a) > 0? (a) : 0;
```

Pero que pasa si pongo esto:

```
auto a = 3.0;
auto result = ReLU(++a);  // Not works result = 5 WTF!!!!
//same as:
auto result = (++a) > 0? (++a) : 0;
```

Así que no, macros no. Te presento `templates` que lo que nos permite es parametrizar una función (en el futuro te enseñare a hacerlo con una clase / struct).

Esto es útil si lo que tenemos son muchas funciones que hacen exactamente lo mismo pero que cambian con el tipo de dato que reciben.

```
template <typename number>
number ReLU(number input) {
    return input > 0? input : 0;
}
```

Para usarlas esta es la sintaxis:

```
int main () {
    int a = 3.5;
    int result1 = ReLU<int>(a);  // complete name
    int result2 = ReLU(a);       // Let the compiler do the work
}
```

Algo importante es que todo este proceso se hace en tiempo de compilación por lo que no hace tu programa mas lento, además tienes al compilador de tu lado, por lo que si tienes un error el compilador puede ayudarte a encontrarlo.

Puedes pensar que lo que hacemos al declarar un `template` es como hacer una plantilla para funciones, ahora cuando hacemos la línea: `int result1 = ReLU<int>(a)` el compilador dice, mira necesitamos una función donde `number = int` , es decir tras bambalinas el compilador crea una función como esta, reemplazando `number` con `int`

```
int ReLU(int input) {
    return input > 0? input : 0;
}
```

Si te das cuenta la sintaxis es bastante sencilla, pero aun asi vamos a explicarla con más detalle, pues fue de las cosas que mas me costo y la verdad es que no hay muchos lenguajes que tienen esta sintaxis.

4.10.1. Sintaxis

Una `template` tiene esta sintaxis:

```
// template a function
template <typename type1, typename type2, typename type3...>
auto someFunction(...) {
    ...
}

// template a class / struct
template <typename type1, typename type2, typename type3...>
class / struct someName {
    ...
}
```

Ahora, ¿Qué es lo que hace esto? Pues lo que permite dependiendo del tipo de dato que necesitemos el compilador generar por nosotros tantas versiones de esta función / programa como necesitemos.

Ahora, la linea `template <typename type1, typename type2, typename type3...>` tiene que ir justo antes del inicio de una función o una clase y solo afectara a es clase o función.

Veamos otro ejemplo:

```
template <typename val>
struct node {
    val data;
    node<val>* next = nullptr;
};

auto main() -> int {
    auto n1 = node<int>{1};
    auto n2 = node<std::string>{"hi"};

    return 0;
}
```

4.10.2. Valores por defecto a la hora de declararlas

Ahora si quieres darles valor por defecto a cosas la sintaxis es bastante sencilla: Ve por ejemplo esta función, no te preocupes por enteder como funciona por dentro o que hace pero si como es que podemos darles valor por defecto:

```
template <typename integer, typename unsignedInteger = unsigned int>
auto binaryExponentation(integer base, unsignedInteger exponent) ->
   integer {
    auto solution = integer {1};

    while (exponent > 0) {
        if (exponent & 1) solution = base * solution;

        base = base * base;
        exponent = exponent >> 1;
    }

    return solution;
}
```

Entonces como ahora tenemos valores por defecto podemos hacer algo como:

```
int main() {
    cout << binaryExponentation<int>(3, 4) << endl;
    return 0;
}
```

Y el compilador entederá que `integer = int` y que `unsignedInteger = unsigned int`

4.10.3. Parametrizar cosas que no son `type`

No solo podemos parametrizar tipos en sí, sino también cualquier variable, por ejemplo:

```
template <typename element, size_t size>
struct myArray {
    element data[size];
    ...
};
```

Por lo que ahora podríamos hacer algo como:

```
int main() {
    myArray<int, 10> a {};
    return 0;
}
```

Que si te das cuenta es mas o menos la misma técnica que usa `std::array` .

4.11. I/O: La entrada / salida en C++

4.11.1. Introducción a `std::cout`

En realidad la entrada y la salida en C++ es bastante sencillo, mira:

```
// In C
#include <stdio.h>

...

int a, b, c, d;
printf("%d %d %d", a, b, c);
```

```
// In C++
#include <iostream>

...

int a, b, c, d;
std::cout << a << b << c;
```

Como te podrás dar cuenta no necesitas indicar el tipo de dato que vamos a imprimir, basta con que sea "imprimible", o mas específicamente que sobrecarge el operador <<.

Tampoco es necesario que todas las cosas que imprimamos a la vez tengan el mismo tipo de dato.

```
auto a = int{3};
auto b = std::string{"hola"};
std::cout << a << b;        // print: "3hola"
std::cout << a << " " << b; // print: "3 hola"
```

Para dar un salto de línea podemos usar:

```
auto a = int{3};
auto b = std::string{"hola"};
std::cout << a << "\n" << b;
/*
    print:
    "3
    hola"
*/
```

Modificadores

Si quieres manipular la forma de imprimir cosas tienes que incluir `iomanip` .

```
#include <iomanip>
#include <iostream>

auto main() -> int {
    auto f = 301.14159;
    std::cout << std::setprecision(5) << f << '\n';
    std::cout << std::setprecision(9) << f << '\n';
    std::cout << std::fixed;
    std::cout << std::setprecision(5) << f << '\n';
    std::cout << std::setprecision(9) << f << '\n';
    return 0;
}
```

Va a imprimir esto:

```
301.14
301.14159
301.14159
301.141590000
```

También puedes hacer esto:

```
#include <iostream>     // std::cout, std::endl
#include <iomanip>      // std::setbase

int main () {
    std::cout << std::setbase(16) << 110 << std::endl;
    return 0;
}
```

Va a imprimir esto:

```
6e
```

[9]

4.11.2. Introducción a `std::cin`

Ahora si quieres leer información hacia tu programa puedes hacer algo como:

```
auto a = int{};
auto b = std::string{};
std::cin >> a >> b;
```

Y lo que leera sera un entero después un espacio o un salto de línea y luego una cadena de texto hasta el siguiente espacio o salto de línea.

Por ejemplo una entrada a este programa podría ser:

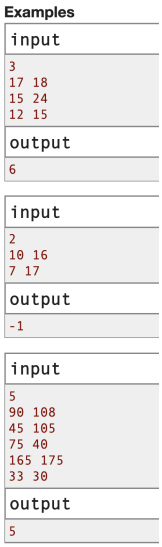
```
0 hola
```

```
-1212 elbebe
```

```
0
leilan
```

Esto es importante, `cin` lee cada variable hasta un salto de línea o espacio.

Generalmente en programación competiva, siempre en la misma entrada te dan cuantas lineas vas a leer, no se si me explico, algo así:



Examples

input
3 17 18 15 24 12 15
output
6

input
2 10 16 7 17
output
-1

input
5 90 108 45 105 75 40 165 175 33 30
output
5

Figura 4.3: Ejemplo de alguna entrada a algun problema, la verdad no importa cual realmente

Entonces podriamos hacer un código como esto:

```cpp
#include <iostream>

auto main() -> int {
    auto infoToRead = int{};
    std::cin >> infoToRead;

    int a, b;
    for (auto i = int{}; i < infoToRead; ++i) {
        std::cin >> a >> b;
        ...
    }

    return 0;
}
```

Pero, ¿Qué es lo que pasa cuando no nos dan eso? Por ejemplo supongamos que tenemos un montón de números:

```
1 2 4 6 -10
```

Podemos usar este código:

```
auto number = int{};
while (std::cin >> number) {
    ...
}
```

Esto funciona porque la expresión `while (std::cin >> number)` regresa `false` solo cuando ya no se puede leer mas.

std::getline

Ahora si lo que queremos es leer línea a línea podemos usar la función: `std::getline`

```
// syntax: std::getline(input, output);

auto name = std::string{};
std::getline(std::cin, name);
```

std::istringstream

Supongamos que tenemos una montón de listas de números por ejemplo:

```
1 2 4
3 4
2
1 3 3 54 12
```

Esta entrada tiene 4 listas:

- 1, 2, 4
- 3, 4
- 2
- 1, 3, 3, 54, 12

Como tienes varias listas no puedes aplicar la de `while (std::cin >> number)` porque aun puede leer más.

Por lo que puedes hacer algo como:

```
#include <iostream>
#include <sstream>
#include <vector>
```

```
auto main() -> int {
auto buffer = std::string {};
while (getline(std::cin, buffer)) {
    auto bufferStream = std::istringstream {std::move(buffer)};

    auto list = std::vector<int> {};
    auto num = int {};
    while (bufferStream >> num) list.push_back(num);

    ... (use list)

}

return 0;
}
```

4.11.3. Hacer la I/O lo más rápido posible

Hay dos grandes cosas que puedes hacer para hacer acelerar el entrada y la salida de datos:

- **Puedes separar la sincronización que esta por defecto**:

  ```
  // No merge cin/cout with scanf/printf
  ios::sync_with_stdio(false);
  ```

  Lo que hace esto es eliminar la sincronización que existe por defecto entre `std::cout / std::cin` y `printf / scanf` .

  Si en tu código no estas usando `std::cout / std::cin` y `printf / scanf` entonces no tendrás ningun problema. Y con esto lograrás una gran ventaja de tiempo.

  Veamos un ejemplo:

  Antes:

  ```
  cout << "Hello";
  printf("World");
  cout << "Ciao";

  // Will always print "HelloWorldCiao"
  ```

  Después:

  ```
  ios::sync_with_stdio(false);

  cout << "Hello";
  printf("World");
  cout << "Ciao";
  ```

```
// Will maybe print:
//    -  "HelloCiaoWorld"
//    -  "HelloWorldCiao"
//    -  "CiaoHelloWorld"
```

Pero recuerda que si siempre usas la entrada y salida de C++ no hay problema:

```
ios::sync_with_stdio(false);

cout << "Hello";
cout << "World";
cout << "Ciao";

// Will always print "HelloWorldCiao"
```

- **Puedes separar la sincronización entre la entrada y la salida**:

  Si tu programa solo leera al principio y después hará cálculos y al final dará un resultado entonces puedes hacer algo como:

  ```
  cin.tie(nullptr);
  ```

  Esto hará algo parecido a lo de arriba pero entre `std::cout` y `std::cin`.

  Más tecnicamente lo que hace es es eliminar esta propiedad que C++ hace por defecto "flushing of std::cout before std::cin accepts an input".

- **Puedes leer enteros de manera aún mas rápida**:

  ```
  template <class number>
  inline auto getNumberFast() -> number {
    auto result = number {};
    auto isNegative = false;
    auto currentDigit = char {getchar_unlocked()};

    while (currentDigit < '0' or currentDigit > '9') {
      currentDigit = getchar_unlocked();
      if (currentDigit == '-') isNegative = true;
    }

    while ('0' <= currentDigit and currentDigit <= '9') {
      result = (result << 3) + (result << 1);
      result += currentDigit - '0';
      currentDigit = getchar_unlocked();
    }

    return isNegative ? -result : result;
  }
  ```

  Puedes usarla de esta manera:

```cpp
#include <vector>

auto numberOfElements = getNumberFast<int>();

std::vector<int> elements {};
elements.resize(numberOfElements);

for (auto& element : elements) element = getNumberFast<int>();
```

- **Quizá debas usar `\n` en vez de `std::endl`**

  Si no necesitas hacer `flush`, entonces segurante no necesites `std::endl`.

  Recuerda que hacer `flush` es tomar la información que esta en un buffer y escribirlo de verdad en la salida estandar, en la terminal o el archivo al que debe de ir.

  Pues por defecto se guarda en un buffet temporal (que es mas rápido) y se guardar en el lugar al que debe de ir al final.

  Generalmente esta optimización es hasta el doble de rápida que `std::endl`.

Capítulo 5

Cosas Avanzadas de C++

5.1. using

Creo que si sabes algo de C entonces conoces typedef, que lo que nos permite es darles nuevos nombres a tipos de datos, por ejemplo:

```
typedef int MyInt;
```

En C++ 11 y en adelante tenemos otra forma de expresar la misma idea, que en lo único que cambia es la sintaxis pero la idea es la misma y creo (personalmente) que es mucho mas entendible:

```
using MyInt = int;
using vi = std::vector<int>;

int main () {
    auto x = MyInt{1}; // x is an int
    auto y = vi{}; // y is an std::vector of ints
}
```

Además `using` es compatible con `templates` logrando hacer cosas como esta:

```
template <typename T>
using matrix = std::vector<std::vector<T>>;

int main () {
    auto m = matrix<int>{}; // m is an vector of vectors of int
}
```

5.2. Sobre inicializar

(AAA) Almost always auto

Desde hace unos años se esta dando una forma estándar de inicializar cosas en C++ que se basa en seguir este patrón:

```
// For declaring types
using T = ...;

// For declaring variables
auto x = ...;
auto y = type{...};
```

Por ejemplo:

```
int i = 42;                              ->   auto i = 42;
long v = 42;                             ->   auto v = long {42};
Costumer c {"Jim"};                      ->   auto c = Costumer {"Jim"};
vector<int>::iterator p = v.cbegin(); ->   auto p = v.cbegin();
```

Un gran problema con esta idea son cosas que son caras de copiar o que no se pueden mover (como por ejemplo), `atomic, array, unique_ptr` . Por lo que usalo con cuidado y en general recomiendo solo usarlo cuando lo estamos declarando variables sencillas y no contenedores, pero no puedo negar que se ve hermoso y tiene un par de ventajas importantes.

5.3. Lifetime: La mejor característica de C++ : }

Se le pregunto a varios grandes programadores de C++ cual era su característica más importante y casi por unanimidad dijeron:

}

Y no, no estoy bromeando, este símbolo no es solo para decirle al compilador que se acaba de terminar el scope actual (es decir, que se acabo la función o el bucle o la condicional o la clase, etc...) sino que es justo en este momento y solo en este momento cuando C++ limpia toda la basura.

Por ejemplo dados estos códigos:

```
int do_work() {
    auto x = ...;
}

...

class shape {
    container points;
}
```

Es justo cuando el compilador ve } que se da la orden de limpiar, en ese momento es cuando se destruye la variable x o cuando destruyes a un objecto de tipo shape automaticamente se destruye el contenedor de puntos.

En otras palabras porque las reglas de C++ sobre el scope de las variables y objectos te da una manera determinista y segura de finalizar weas.

Es una forma automática y segura de liberar recursos cuando ya no los estoy usando.

En otras palabras, la vida o lifetime de un objecto esta atada a su scope . Y como me gusta decirlo: **Todo el C++ es la responsabilidad de alguien.**

O en inglés: **Everything is owned by someone.**

[2]

5.4. const

Que algo sea `const` es una forma de expresar en nuestro código que algo no se puede cambiar.

Dependiendo de donde se ponga significará una cosa u otra:

- **const variable**

```
const int days {10};
days = 20;              // Illegal: days cannot change
```

- **const reference**

```
struct point {int x, y};

point p1 {1, 2};
const point& p2 {p1};
p1.x = 20;                    // Illegal: point cannot change
```

- **const pointer**

```
struct point {int x, y};

point p1 {1, 2}, p2 {2, 3};
point* const p3 {&p1};

p3 = &p2;                   // Illegal: p3 cannot change
```

- **pointer to const**

```
struct point {int x, y};

const point p1 {1, 2}, p2 {2, 3};
const point* p3 {&p1};

p3 = &p2;                   // Ok
p3.x = 30;                  //Illegal: p3 cannot change
```

- **const method**

```
struct point {
    int x, y;

    // Ok print do not change anything
    auto print() const {
        cout << x << ", " << y << '\n';
    }

    // Illegal: print change something, so no const
    auto print2() const {
        x = 20;
        cout << x << ", " << y << '\n';
    }
};
```

5.5. lvalues vs rvalues categories

En C++ todas las "cosas" pueden clasificarse de dos manera:

- **lvalues**:

  Son cosas que hacen referencia (en el sentido mas general de la palabra) a "algo" que tiene una localización en memoria.

  Otra forma de decirlo es que los `lvalues` son las cosas a las que puedes sacarle la dirección de memoria con el operador `&` .

  En nombre venia originalmente de que podiamos ver los `lvalues` como las cosas que podriamos poner a a izquierda de una asignación.

  Por ejemplo:

```
int days = 10;                  // days: lvalue
int& referenceToDays = days;    // referenceToDays: lvalue
const Widget x {};              // x: lvalue
Widget* button = new Widget{};  // button: lvalue
```

- **rvalues**:

  Son todas las cosas que no tienen una localización (y por lo tanto tampoco una dirección) en memoria, generalmente son cosas como temporales o constantes.

  Por ejemplo:

```
int days = 10;                  // 10 is an rvalue
int& referenceToDays = days;    // here days is an rvalue
widget x = widget {};           // widget {} is an rvalue
```

  [4]

5.6. References types: Tipos de referencias

Veamos todas las posibles referencias con un ejemplo sencillo, a través de una función: La primera forma y sin involucrar referencias sería pasar un parámetro por valor.

Por ejemplo, dada la función

```
void func(Widget pb);
Widget x;

func(x);                //call with an lvalue: valid :)
func(Widget {});        //call with an rvalue: valid :)
```

5.6.1. lvalue references

Son de las que habíamos hablado antes, son formalmente conocidas como `lvalue references` y podemos ver un ejemplo de ellas aquí:

```
void func(Widget& pb);
Widget x;

func(x);                //call with an lvalue: valid :)
func(Widget {});        //call with an rvalue: ERROR :(
```

Esto último no es válido porque cuando aceptamos un parámetro por una referencia normal lo que estamos haciendo es una promesa: prometemos que quien haya llamado a `func` podrá ver los cambios que realiza la función.

Pero al llamarla con un temporal es imposible ver los cambios que realizaría, por eso no es válido.

Este tipo de referencias estan diciendonos que podemos modificar la información que entre a la función y que la persona que llamó a la función podrá ver los cambios.

5.6.2. const lvalue references

Podemos también usar este tipo de referencias y ve como soluciona el problema:

```
void func(const Widget& pb);
Widget x;

func(x);                //call with an lvalue: valid :)
func(Widget {});        //call with an rvalue: valid :)
```

Esto último es válido porque cuando aceptamos un parámetro por una referencia constante estamos haciendo una promesa, pero una diferente, decimos que necesitamos la información del objeto real, que no necesitamos una copia, pero que no vamos a realizar

cambios, por lo tanto como no estamos prometiendo que la persona que llame a `func` pueda ver cambios (porque no habrá) entonces no hay problema con los temporales.

Este tipo de referencias estan diciendonos que NO podemos modificar la información que entre a la función y que por lo tanto da lo mismo si la persona que llamó a la función puede o no ver los cambios.

5.6.3. rvalue references

Podemos también ver que existe este otro tipo de referencias, estas no se unen a variables, es decir a `lvalues` , sino a `rvalues` es decir se une a temporales.

Este tipo de referencias son especiales porque solo se pueden hacer a temporales. Veamos como sigue el ejemplo:

```
void func(Widget&& pb);
Widget x;

func(x);                //call with an lvalue: ERROR :(
func(Widget {});        //call with an rvalue: valid :)
```

La promesa con este tipo de refencias es que tendremos el objecto real, no copias pero que quien llame a la función no podrá ver los cambios que hemos hecho.

Este tipo de referencias estan diciendonos que podemos modificar la información que entre a la función, pero que la persona que llamó a la función no podrá, por lo tanto solo acepta temporales.

[4]

Otro ejemplo de estas referencias podría ser esto:

```
int add(int a, int b) { return a + b; }

int result   = add(4, 5);         //Ok :)
int& result2 = add(4, 5);         //No Ok :(
```

El error con el siguiente código es que `result2` es una referencia y no tiene a que unirse porque el resultado de `add(4, 5)` es un temporal, por lo tanto da un error; para eso estan estas nuevas referencias.

```
int add(int a, int b) { return a + b; }

int result    = add(4, 5);         //Ok :)
int&& result2 = add(4, 5);         //Ok :)
```

Algo interesante es que estas referencias no son constantes, es decir podemos hacer algo como:

```
int add( int a, int b) { return a + b; }

int&& result = add(4, 5);                  //Ok :)
// address of result: 0x7ffe79121f8c

result = add(5, 7);                        //Ok :)
// address of result: 0x7ffe79121f8c
```

[5]

5.6.4. Universal References / Fowarding References

Estas dos funciones parecen hacer casi lo mismo:

```
void foo(int&& i);

template <typename T>
void bar(T&& i);
```

Pero estan haciendo cosas algo diferentes, porque pasa esto:

```
int number {};

foo(3);                  //Ok called with an rvalue
foo(number);             //Error: called with an lvalue

bar(3);                  //Ok called with an rvalue
bar(number);             //Ok ... wait WTF!!!!!!
```

La cosa esta en que `T&&` es un tipo especial de referencia que se puede unir tanto a `rvalues` como a `lvalues` .

Y por así decirlo van a transformarse en referencias a `lvalues` o a `rvalues` dependiendo de que le pasemos, si una variable o un temporal respectivamente.

Este tipo de refencias solo existen dentro de `templates` o usando `auto&&` , y por muy poco usadas, sobretodo en programación competitiva, excepto por los `for range` , y hablaremos pronto de porque este tipo de referencias son muy útiles para esos casos.

Una frase que me gusta es esta: `Using auto&& or universal references has the advantage that you captures what you get` .

Algo que hay que tener en cuenta es que añadir `const` vuelve a hacer que `&&` funcione como una `rvalues reference` .

Ve este ejemplo:

```
template <typename T>
void bar(const T&& i);

int number {};
```

```
bar(3);                 //Ok called with an rvalue
bar(number);            //Error: called with an lvalue
```

5.6.5. auto vs auto& vs const auto& vs auto&&

`auto` , el clásico `auto` . Hay que tener algo de cuidado con lo que hacemos con el, hay que entender que pasa en ciertos casos especiales con el `auto` .

Usar algo como:

```
auto someValue {...};
```

Solo nos dice, compilador, pon el tipo de dato que creas correcto aquí.

Además hay algo importante que tienes que recordar:

auto decay

Supongamos que haces algo como:

```
auto var1 = var2;
```

Uno pensaría que `var1` y `var2` , tendrían el mismo tipo, y esto es casi siempre así, pero `auto` sigue además otras reglas:

- Los arreglos estilo C (`int []`) se vuelven punteros (`int*`)
- Las funciones se convierten en punteros a funciones
- Se eliminan las referencias top-level
- Se eliminan los top-level `const` y `volatile`

Así por ejemplo queda claro porque pasa esto:

```
int var1 {3};
const int& var2 = var1;      // var2 is an const int&
auto var3 = var2;            // var3 is just an int

auto s = "hi"                // s has type const char*
                             // but "hi" has type const char[3]
```

Este último ejemplo explica que es un `top-level` , osea si tenemos una variable que es `const` entonces por ejemplo una referencia constante, esta será eliminada, pero si tenemos por ejemplo un arreglo de carácteres constantes entonces seguiremos teniendo el `const` , tendremos un puntero a carácteres constantes.

En otras palabra un modificador es `top-level` es aquel que afecta a la variable en si y no a las cosas que apunto o que hace referencia, osea el `const` de un puntero constante es `top-level` pero el `const` de un puntero a algo constante NO es `top-level` .

Entonces, en resumen poner `auto` no hace nada relacionado con `const` o sobre la creación de las referencias.

Modificadores del `auto`

- `const auto`

  Lo que nos dice esto es que vamos a crear una variable del tipo que el compilador crea que sea correcta, pero que ademas esta variable será `const` , por ejemplo:

```
auto coins {10};            //Equals: int coins {10};
const auto coins {10};      //Equals: const int coins {10};
```

```
vector<const int> numbers {1, 2, 3};
for (const auto num : numbers) {        // num is const int
    cout << num << '\n';
}
```

- `auto&`

  Lo que nos dice esto es que vamos a crear una referencia al tipo de dato que el compilador sabe que es el correcto, por ejemplo:

```
vector<const int> numbers {1, 2, 3};
for (auto& num : numbers) {        // num is int&
    cout << num << '\n';
}
```

```
auto coins {10};            //Equals: int coins {10};
auto& coins2 {coins};       //Equals: int& coins2 {coins};
```

- `const auto&`

  Hace lo que esperas, una referencia constante al tipo de dato correcto.

- `auto&&`

  Esta es interesante, lo que nos genera es un `fowarding / universal reference` al tipo de dato correcto, hablamos de ella hace poco, es lo que nos da es una forma de tener referencias a `lvalues` o `rvalues` , pero creo que queda claro con un ejemplo:

```
int numberOfDays {10};
auto&  ref1 {numberOfDays};  // ref1 is an int&
auto&& ref2 {5};             // ref2 is an int&&
auto&& ref3 {numberOfDays};  // ref3 is an int&
```

5.6.6. `for (auto& x : v)` vs `for (auto&& x : v)`

Supongamos que tenemos algunos contenedores como:

```
vector<int> numbers {1, 2, 3, 4, 5};
vector<bool> flags {true, true, false, true, true, false};
vector<int> words {"hello", "world", ":D"};
```

Entonces podemos hacer algo como:

```
for (auto num : numbers) {
    cout << num << '\n';        // Will work [1, 2, 3, 4, 5]
}
```

Ahora, esto es bastante ineficiente porque en cada paso estamos haciendo copias por lo que si solo queremos mirar a la información que esta dentro entonces podemos hacer algo como:

```
for (const auto& word : words) {
    cout << word << '\n';    // Will work ["hello", "world", ":D"]
}
```

Si queremos editar estos entonces podemos simplemente quitar el `const` :

```
for (auto& num : numbers) {
    num += 1;               // Will work [2, 3, 4, 5, 6]
}
```

El único problema son algo llamado los `proxy-iterators` (como por ejemplo en un `vector<bool>`), por lo tanto si quiere evitar problemas podemos simplificar las cosas y decir:

- Para ver los objectos

  ```
  for (const auto& num : numbers)
  ```

- Para editar los objectos

  ```
  for (auto&& flag : flags)
  ```

[7]

5.7. Enums

5.8. Semantics

Una semantica es una forma programar, es algo parecido a un paradigma.

Siendo mas formales una semantica en este contexto es hablar como es que un tipo de dato (predefinido o creado por el programador) será usado.

5.8.1. Value Semantics

Son aquellas que estan pensadas para ser copiadas, es decir son aquellas en la que usas el operador `=` para cambiar le valor de la variable, generalmente son lo que llamamos inmutables, es decir que contienen información (o en otras palabras un estado) que no puede cambiar una vez han sido creadas.

Por ejemplo:

```
int someNumber {3};

// If I want to change the value I only have the option to use =
someNumber = 4;

// The next is possible, not super OK, but compiles
string welcomeText {"Hello"};
welcomeText = "Hello :D";
```

5.8.2. Move Semantics

Quiza una pregunta que tengas es:

Si tu me dijiste que C++ por defecto manera que sus variables son valores entonces tendrás que hacer un monton de cosas para evitar andar creando objectos (que a veces pueden ser enorme como una colección por ejemplo) temporales y luego copiando toda su información, eso suena a algo muy costoso.

Por ejemplo, cuando tenemos algo como:

```
auto getMeNumbers() -> vector<int> {
    vector<int> numbers {1, 2, 3, 4, 5, 6};
    return numbers;
}

auto result = getMeNumbers();
```

Tu podrías preguntarte algo como: *¿No es muy costoso andar copiando estos vectores?*

Y debido a eso C++ 11 introdujo algo que se conoce como move semantics.

Y se basa en la idea de que si tienes una cosa muy compleja o enorme que esta apunto de ser destruida (como por ejemplo `numbers`) en vez de copiar elemento a elemento la información a un nuevo vector (para crear el valor que vamos a retornar de la función) para luego destruir el vector original; ¿Porqué no mejor tomar *responsabilidad* del objecto inicial, como dice mucha gente, tomar sus entrañas (generalmente asignando un par de punteros) pues al fin y al cabo es un objecto que esta a punto de ser destruido.

A esta operación le llamamos mover un objecto.

[2]

Y es lo que hace por defecto `C++` cuando regresas de una función una cosa que implementa `move semantics` , por ejemplo `std::vector` la implementa, por lo tanto tu no tienes que preocuparte de eso, asi que si, es bastante barato regresar por valor incluso cosas que soy muy complejas y tardadas de copiar, porque nunca las estas copiando, las estas moviendo.

Algo importante es que al objecto del que movemos la información estará después del `move` en un estado *zombie*, en el que ya no podemos confiar en su contenido, por eso es que se basa en los `rvalues references` .

Una `rvalues reference` es una variable que hace referencia a un objecto que esta siendo usado como un temporal, es decir, es un objecto que esta a punto de ser destruido.

Parte III

Estructuras de Datos básicas y los containers en C++

Capítulo 6

Introducción

6.1. Los contendedores en si

Capítulo 7

Arrays

Un array es la estructura de datos mas sencilla que hay, pero quizá la más útil de todas: es la gran olvidada e infravalorada.

7.1. Una idea abstracta

Un `array` es un conjunto de elementos del mismo tipo, contiguo en memoria, al ser contiguo la idea para acceder a los elementos es decir, dame el 1 elemento, dame el 5 elemento, dame el 10 elemento.

7.2. std::array

En `C++` si quieres crear un arreglo de elementos contiguo y **conoces en tiempo de compilación el tamaño de dicho arreglo** entonces puedes usar `std::array` .

7.2.1. Inicialización

- Si quieres declarar un array, es super sencillo:

```
std::array<type, sizeOfTheArray> arrayName;
```

- Se parece a los arreglos de C, pero mejor.

  Nota que `array` , se comporta de una manera parecida a los arrays de C:

```
array<int, 4> a1;             // Ok, 4 elements, all are undefined
array<int, 4> a2 {};          // Ok, 4 elements: [0, 0, 0, 0]
array<int, 4> a3 {1, 2};      // Ok, 4 elements: [1, 2, 0, 0]
```

  Pero ten cuidado al inicializar un array de "cosas" vas a necesitar un par de parentesis más:

```
using complex = std::complex<int>
vector<complex> v1 {{1, 2}, {3, 4}};
// Ok, 2 elements: [1+2i, 3+4i]

array<complex, 2> a2  {{1, 2}, {3, 4}};
// Syntax error

array<complex, 2> a2  {{{1, 2}, {3, 4}}};
// Ok, 2 elements: [1+2i, 3+4i]
```

  Así que en resumen ten cuidado, necesitas un par mas de `{ }` cuando y solo cuando estas inicializando cosas anidadamente.

7.3. std::vector

7.3.1. Inicialización

Hay dos formas de inicializar un vector:

- Iniciar un vector con un tamaño dado y un valor por defecto a cada elemento

```
// vector with 7 ints, all start with 0
vector<int> days (7, 0);

// vector with 13 strings, all are "" for now
vector<string> names (13, "");
```

- Iniciar un vector con unos elementos:

```
// vector with 5 ints: [2, 3, 5, 7, 11]
vector<int> primes {2, 3, 5, 7, 11}

// vector with 2 strings: ["Oscar", "Rosas"]
vector<string> names {"Oscar", "Rosas"}
```

7.3.2. Como es que logra autodimensionarse

7.3.3. Cosas que puedes hacer

-

7.3.4. Tips

- Si quieres hacer que sea mas rápido y conoces cual será su tamaño máximo, entonces puedes prereservar el espacio máximo, esto evitará que entre aumentos de tamaños se tengan que mover los elementos, haciendo el código un poco más rápido.

  Se ve así:

```
#include <vector>

int numberOfElements;
cin >> numberOfElements;

std::vector<int> elements {};
elements.resize(numberOfElements);

for (auto& element : elements) cin >> element;
```

7.4. Algunos problemas clásicos

- Encontrar el segundo menor elemento en un arreglo:

```
#include <array>
#include <iostream>
#include <vector>

template <typename container>
auto findSecond(const container& data) {
  // We need 2 elements to find the second min :c
  assert(data.size() > 1);

  auto minimum = data[0], almostMin = data[1];
  for (const auto current : data) {
    // If we have found a new min move everything
    if (current < minimum) {
      almostMin = minimum;
      minimum = current;
      // Ok, not new min, but maybe we found a new 2 min
    } else if (current < almostMin)
      almostMin = current;
  }
  return almostMin;
};

auto main() -> int {
  using namespace std;
  using number = const int;

  vector<number> data1 {1, 4, 1};
  cout << findSecond(data1) << endl;  // 1

  array<number, 5> data2 {1, 4, -1, 3, -5};
  cout << findSecond(data2) << endl;  // -1

  return 0;
}
```

- Encontrar el primer elemento que no se repite en un arreglo:

```
#include <iostream>
#include <unordered_map>
#include <vector>

template <typename container>
auto findSecond(const container& data) {
  using element = typename container::value_type;

  enum class found : char { one_time, more_than_1_time };
  auto state = std::unordered_map<element, found> {};
```

```
  for (const auto val : data) {
    auto situation = state.find(val);
    if (situation == end(state)) {
      state.insert({val, found::one_time});
    } else if (situation->second == found::one_time)
      situation->second = found::more_than_1_time;
  }

  for (const auto val : data) {
    auto situation = state.find(val);
    if (
      situation != end(state) and
      situation->second == found::one_time) {
      return val;
    }
  }

  // Maybe throw or something
  return element {};
};

auto main() -> int {
  using namespace std;
  using number = int;

  vector<number> data1 {1, 4, 1, 0};
  cout << findSecond(data1) << endl;  // 1

  return 0;
}
```

7.5. std::string

7.6. std::bitset

7.7. Si quieres un programa rápido: Verdaderos arrays

Capítulo 8

Contenedores heteromorfos

8.1. std::pair

8.2. std::tuple

Capítulo 9

Contenedores asociativos

9.1. std::map

9.1.1. Inicialización

Puedes inicializar usando una `ininitializer_list` , se ve mucho mas claro:

```
// Before :c
map<int, string> translate {};

translate.insert(make_pair(0, "null"));
translate.insert(make_pair(8, "eight"));
translate.insert(make_pair(15, "fifteen"));

// After :D
map<int, string> someMap {
    {0, "null"},
    {8, "eight"},
    {15, "fifteen"},
};
```

9.1.2. Tips

- Prefiere usar `auto` en vez de decir el tipo al iterar sobre un map. Me explico:

  ```
  map<int, string> someMap {
      {0, "null"},
      {8, "eight"},
      {15, "fifteen"},
  };

  // Good: No element copy
  for (const auto& element : someMap) { ... }

  // Bad: Copies element
  for (const pair<int, string>& element : someMap) {...}

  // Ok, but impossible to remember: No element copy
  for (const pair<const int, string>& element : someMap) {...}
  ```

  Y si no me creen vayan a aquí o aquí.

  [8]

- Ten cuidado al usar `map operator[]` ve este ejemplo:

  ```
  map<int, string> someMap {
      {0, "null"},
      {8, "eight"},
      {15, "fifteen"},
  };
  ```

```
if (someMap[9] == "nine") {
    ...
}
```

Lo que acaba de pasar es que si existia el elemento con llave 9 existia pues hacer `someMap[9]` nos lo regresa, pero si no existia acaba de crearlo con un valor por defecto, lo cual esta muy pero muy raro.

Si quieres buscar en un contenedor así puedes usar:

```
if (someMap.find(9) == "nine") {
    ...
}
```

O si solo quieres saber si existe puedes hacer algo como:

```
if (someMap.find(9) != someMap.end()) {
    ...
}
```

O usar esto:

```
if (someMap.count(9)) {
    ...
}
```

- Conservan el ordén:

```
auto someMap = map<int, string> {
    {15, "fifteen"},
    {0, "null"},
    {8, "eight"},
};

for (const auto& p : someMap) cout << p.first << '\n';
// Will print 0, 8, 15
```

9.1.3. std::unordered_map

Capítulo 10

Contenedores únicos

10.1. std::set

- Conserva el ordén

10.1.1. std::unordered_set

Capítulo 11

Contenedores de orden

11.1. Stacks LIFO: Pilas

11.2. Queue FIFO: Colas

11.3. Dequeue: ¿Porqué no las dos?

11.4. Heap

Parte IV

Ideas de las Ciencias de la Computación

Capítulo 12

La Complejidad

12.1. Cotas y Notaciones

12.1.1. Big O Notation: Notación de O grande

Capítulo 13

Optimización

13.1. Límites de Tiempo de Ejecución

13.2. Límites de Memoria

Capítulo 14

Problemas NP

Parte V

Algoritmos Generales y std::algorithms

Capítulo 15

No lo hagas tu todo desde cero: std::algorithms

Capítulo 16

Search: Búsquedas

16.1. Linear Search: Búsqueda Lineal

16.2. Binary Search: Búsqueda Binaria

16.3. Ternary Search: Búsqueda Ternaria

16.4. Upper Bound

16.5. Lower Bound

Capítulo 17

Sorting: Ordenamiento por Comparaciones

17.1. Merge Sort

17.2. Quick Sort

17.3. Stable Sort

17.4. Partial Sort

Capítulo 18

Sorting: Ordenamiento NO por Comparaciones

18.1. Bucket Sort

Parte VI

Técnicas de Solución

Capítulo 19

Ad-Hoc

Capítulo 20

Recursividad y BackTracking

Capítulo 21

Divide and Conquer: Divide y Vencerás

Capítulo 22

Greedy

Capítulo 23

Programación Dinámica

Parte VII

Programación es solo matemáticas aplicadas

Capítulo 24

Binary: Explotando el Binario

24.1. Bits

24.1.1. Introducción

24.1.2. Manejo de Bits: Utilidades que podemos hacer con ellos

- Par e impar:

  Con esto también se explica una forma que hacemos para saber si un número es par o impar:

```
if (number & 1) {
    // odd
}
else {
    // even
}
```

  Esto es fácil de enteder porque `number & 1` lo que hace es comparar el último bit que si recuerdas la representación binaria nos dice si un número necesita un $+1$ al final o no y recuerda que solo necesita el $+1$ si es que es impar.

- Dividir (con piso) o multiplicar por potencias de dos muy muy rápido:

  Recuerda la siguiente idea, que es bastante fácil de ver porque funciona si recordamos el binario:

$$x << y = x * 2^y$$
$$x >> y = \left\lfloor \frac{x}{2^y} \right\rfloor$$

Mascara de bits

24.2. Conversiones entre Sistemas (HEX, OCT)

24.3. Binary Exponentiation: Exponenciacion Binaria

24.4. Binary Multiplication: Multiplicación Binaria

Capítulo 25

Hash y manejo de strings

Capítulo 26

Roots: Encontrar Raíces de ecuaciones

26.1. Newton - Ranphson

Capítulo 27

Teoría de Números

27.1. Divisibilidad

27.1.1. Euclides

27.1.2. Criterios de Divisibilidad

27.2. Modulos

27.3. Fibonacci

27.4. Números de Catalán

27.5. Primos y Factores

27.5.1. Eratosthenes Sieve: Criba de Eratóstenes

27.5.2. Prime Factorization: Factorización

27.5.3. Divisores

27.5.4. Euler Totient: La Phi de Euler

27.6. Factoriales

27.7. Digit Sum

Capítulo 28

Probabilidad

28.1. Inclusión Exclusión

28.2. Permutaciones y combinaciones

Capítulo 29

Cálculo

29.1. Series aritmética y geométricas

Capítulo 30

Geometría

Parte VIII

Estructuras de Datos Avanzadas

Capítulo 31

Pointers: Apuntadores

31.1. Pointers (C Style)

No me pondre ahora a explicar en gran medida el concepto de punteros, primero que nada porque es un concepto de gran importancia, así que si algo de lo que digo no te queda completamente claro, no te preocupes. No es fácil entenderlos la primera vez.

Los punteros son como su nombre lo indica variables que "apuntan" a un espacio en memoria. Además recordemos que toda variable en C++ es un fragmento de la memoria, y por lo tanto tiene una dirección donde dicha variable esta guardada en memoria.

Si quieres obtener la dirección de memoria de una variable basta con hacer algo como:

```
int someNumber {3};

// Give us the direction of someNumber
&someNumber;
```

Los punteros son variables capaces y diseñadas para guardar y manipular dichas direcciones. Ahora, como sabes, un `int` y un `long long` no ocupan el mismo espacio en memoria por lo tanto para cada tipo de dato en C++ tenemos un tipo de puntero.

```
int someNumber {3};
int* somePointerToAnInt = &someNumber;

string someText {"Hi"};
string* somePointerToAnString = &someText;

double someDecimal {2.123};
double* somePointerToAnDouble = &someDecimal;
```

31.2. Smart pointers

Capítulo 32

Tipos de datos definidos por nosotros

“ Cuando llego Simula (lenguaje de programación) y la programación orientada a objectos, empezamos a cambiar la forma de pensar:

En vez de enfocarnos en meter todo lo que una persona podría desear dentro del lenguaje, podemos darle la capacidad de crear sus propias abstracciones.

Por lo que si necesitas un vector, creas un vector, si necesitas matrices de 3 dimensiones, puedes crearla, si necesitas hacer registros de empleados puedes hacerlo.

De ahi nació la idea de clases. ”

32.1. class & structs

32.2. RAII

32.3. const methods

32.4. Declaración e implementación separadas

32.5. Template Specialization

Capítulo 33

Linked Lists: Listas Enlazadas

Capítulo 34

Binary Trees: Arboles Binarios

34.1. BTS: Arboles de Búsqueda

34.2. AVL - RedBlackTree: Arboles Autobalanceables

34.3. Trie

34.4. LCA: Last common ancestor

Capítulo 35

Segment Tree

35.1. Ideas principales

35.2. FenwickTree / BIT: Binary Indexed Tree

35.3. Problema de motivación: Prefix sums with update

35.4. LSB

Definimos al $lsb(a)$ como el bit menos significativo del número en binario, por lo tanto podemos definirla como: $lsb(a) := a\& - a$, suponiendo que los negativos en nuestra computadora esten programados por complemento a dos.

Parte IX

Grafos y Flujos

Capítulo 36

Grafos y Gráficas

36.1. Representaciones

36.2. BFS: Breadth-first Search

36.3. DFS: Depth-first Search

36.4. Dijsktra: Camino más cercano

36.5. Union Find / DSU: Disjoint Set Union

Bibliografía

[1] Competitive Programming 3, *Halim and Halim, 2013.*

[2] Build Conference: Modern C++ what you need to know, *Herb Sutter, 2014.*

[3] https://www.quora.com/What-is-your-best-description-of-pointers-in-C-C++-programming

Graham Cox, 2017.

[4] https://www.youtube.com/channel/UC-lNlWEq0kpMcThO-I81ZdQ

CopperSpice, 2019.

[5] https://www.quora.com/What-are-some-practical-use-cases-of-rvalue-references-in-C++

Chris Reid, 2010.

[6] https://shaharmike.com/cpp/lambdas-and-functions/

[7] https://stackoverflow.com/questions/15927033/what-is-the-correct-way-of-using-c11s-range-based-for

[8] https://twitter.com/NicoJosuttis

Nicolai Josuttis.

[9] https://en.cppreference.com/