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 la 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`
- `Reactores - OmegaUp`

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.

- **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.

- **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.

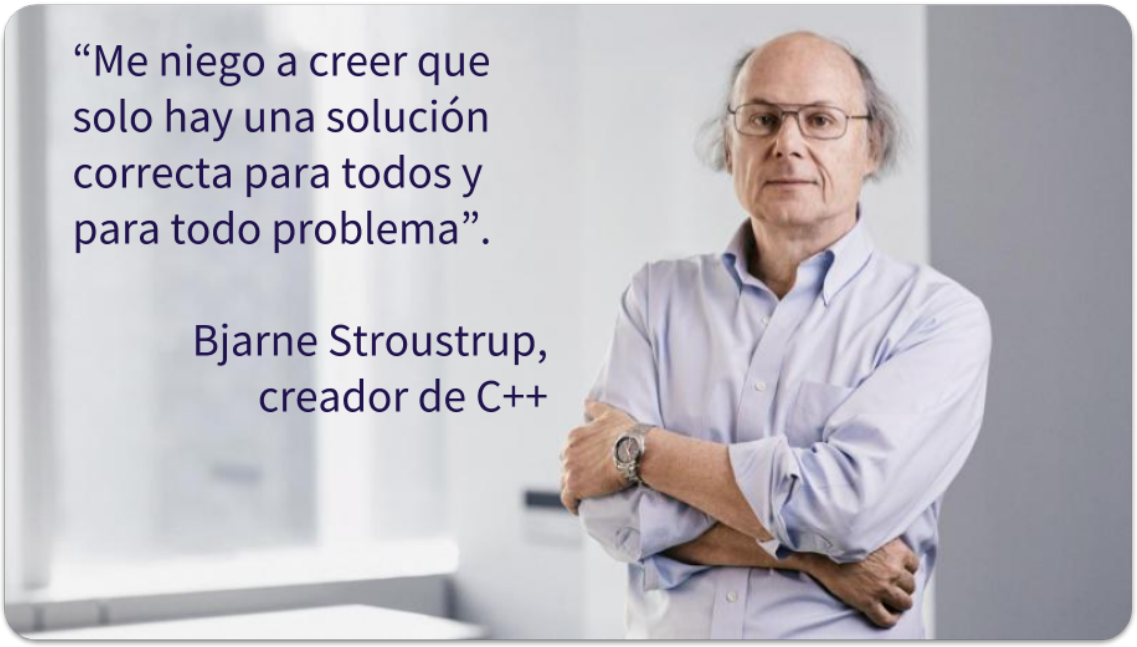
- **El corazón del problema está relacionado siempre con matemáticas, lógica o ciencias de la computación.**

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++`

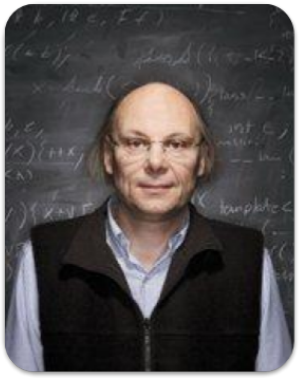


Figura 2.1: Este es el creador del lenguaje (reto: decir su nombre en voz alta) Bjarne Stroustrup

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 estandar, 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 basados 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.

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

  Es decir, con ejemplos en JavaScript tenemos algo como:

  ```
  const person = { month: 3, day: 21 }
  ```

  Resulta que `person` almacena una referencia a ese objecto y no a ese objecto en si, cosa que no pasa en C++ :

  ```
  const map<string, int> person { {"month", 3}, {"day", 21} };
  ```

  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" << endl;
}

// ...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);
vector<shape*> v = load_shapes();

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

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

Bases del Lenguaje

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++` .

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

3.1. 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...

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.

3.1.1. 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 IWantHelp;
```

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 se hace hastante similar:

```
auto someNumber = 20;         //someNumber is int
auto someText = "Hi baby";    //someText is const char* (this is sad)
auto myLoveForYou;            //This will fail
```

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 Inizialization: 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};
auto someText {"Hi baby"};
// this call a someClass constructor
someClass object {"some parameter", someNumber};
```

3.2. Tipos de datos numéricos

3.2.1. 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!

// 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};
```

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 duplicar el tamaño de un int (suponiendo que int tenga 4 bytes de tamaño, y creeme, seguramente es así).

```
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"};
int minValue {-9,223,372,036,854,775,808};
int 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.

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 {};
```

3.2.2. 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.

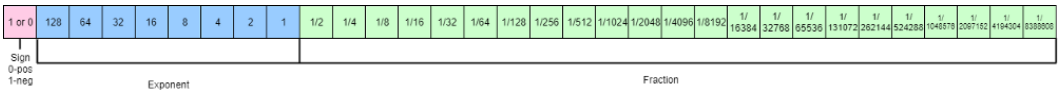


Figura 3.1: Representación del `float` en `C++`

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};
```

3.2.3. 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.

3.3. Los otros tipos de datos fundamentales

3.3.1. void

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

3.3.2. 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};
```

3.4. Operadores

3.4.1. Operadores Aritméticos

Sección corta, si sabes programar los has usado:

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

Ahora los interesantes son los siguientes:

3.4.2. División entera

Una cosa que a mucha gente le cuesta es la división y no porque sea una operación difícil sino porque en `C++` dependiendo de los valores a los que se la apliquemos se comportara de manera o de otra, si es que ambos valores son punto flotante entonces hara lo que esperas por ejemplo `1.0 / 2.0 = 0.5`, pero `1 / 2 != 0.5` porque si aplicas la división a dos enteros entonces hara la división entera, es decir va a redondeo al entero proximo mas pequeño.

Hay que tener cuidado con eso.

3.4.3. 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 $n\,\%k == 0$.

3.4.4. Operadores Lógicos

3.5. Sentencias de Control

3.5.1. 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" << endl;
}

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

  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.

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
if (language == "php") {
    programmerFeelings = "sad";
}
else {
    programmerFeelings = "happy";
}

// The same thing using ternary operator
happyProgrammer = language == "php" ? "sad" : "happy";
```

3.6. Ciclos

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

3.6.1. 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.

3.6.2. 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:

3.6.3. `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 << endl;
}
```

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

3.7. References and value types: Referencias o Valores

3.7.1. 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".

3.7.2. References: Referencias

Una referencia

3.7.3. Pointers: Apuntadores

3.8. 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.

```
    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.

Si te das cuenta no es muy diferente de como

3.8.1. Recursion

3.8.2. Lamdas

3.8.3. Templates

3.9. La entrada / salida en C++

Capítulo 4

Containers: Contenedores de la STD

4.1. std::vector

4.1.1. std::array

4.2. std::string

4.3. std::map

4.4. std::set

Capítulo 5

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

Capítulo 6

Clases: OPP / POO

Capítulo 7

Cosas Avanzadas de C++

7.1. 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]

7.2. Move Semantics: Semanticas de Movimiento

En los viejos tiempo de C++ (y una de las razones por las que mucha gente cree que el lenguaje es muy complejo) es porque la gente se preguntaba lo siguiente:

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

Y debido a eso C++ 11 introdujo algo que se conoce como move semantics y esto es la idea de que si tienes una cosa muy compleja o muy enorme y tu la mueves (como por ejemplo los valores que te regresa una función) entonces C++ no simplemente la copia toda completa sino que toma ownership o responsabilidad de sus entrañas (generalmente asignando un par de punteros) y deja ir al otro (ahora vacío) objecto listo para desaparecer.

[2]

7.3. Si quieres un programa rápido: Verdaderos arrays

7.4. for (auto x : v) vs for (auto& x : v) vs for (auto&& x : v)

Parte III

Ideas de las Ciencias de la Computación

Capítulo 8

La Complejidad

8.1. Cotas y Notaciones

8.1.1. Big O Notation: Notación de O grande

Capítulo 9

Optimización

9.1. Límites de Tiempo de Ejecución

9.2. Límites de Memoria

Capítulo 10

Problemas NP

Parte IV

Estructuras de Datos

Capítulo 11

Arrays

Capítulo 12

Stacks LIFO: Pilas

Capítulo 13

Queue FIFO: Colas

Capítulo 14

Linked Lists: Listas Enlazadas

Una implementación mas o

Capítulo 15

Binary Trees: Arboles Binarios

15.1. BTS: Arboles de Búsqueda

15.2. AVL - RedBlackTree: Arboles Autobalanceables

15.3. Trie

Capítulo 16

Heaps

Capítulo 17

Hash Tables

Parte V

Algoritmos Generales

Capítulo 18

Search: Búsquedas

18.1. Linear Search: Búsqueda Lineal

18.2. Binary Search: Búsqueda Binaria

18.3. Ternary Search: Búsqueda Ternaria

18.4. Upper Bound

18.5. Lower Bound

Capítulo 19

Sorting: Ordenamiento por Comparaciones

19.1. Bubble Sort

19.2. Selection Sort

19.3. Merge Sort

19.4. Quick Sort

Capítulo 20

Sorting: Ordenamiento NO por Comparaciones

20.1. Bucket Sort

Parte VI

Programación es solo matemáticas aplicadas

Capítulo 21

Binary: Explotando el Binario

21.1. Bits

21.1.1. Manejo de Bits

21.1.2. Operaciones con Bits

21.2. Conversiones entre Sistemas

21.3. Binary Exponentiation: Exponenciacion Binaria

21.4. Binary Multiplication: Multiplicación Binaria

Capítulo 22

Roots: Encontrar Raíces de ecuaciones

22.1. Newton - Ranphson

Capítulo 23

Teoría de Números

23.1. Divisibilidad

23.1.1. Euclides

23.2. Modulos

23.3. Fibonacci

23.4. Números de Catalán

23.5. Primos y Factores

23.5.1. Eratosthenes Sieve: Criba de Eratóstenes

23.5.2. Prime Factorization: Factorización

23.5.3. Divisores

23.5.4. Euler Totient: La Phi de Euler

Capítulo 24

Probabilidad

24.1. Inclusión Exclusión

Capítulo 25

Geometría

Parte VII

Técnicas de Solución

Capítulo 26

Ad-Hoc

Capítulo 27

Recursividad y BackTracking

Capítulo 28

Divide and Conquer: Divide y Vencerás

Capítulo 29

Greedy

Capítulo 30

Programación Dinámica

Parte VIII

Grafos y Flujos

Capítulo 31

Grafos y Gráficas

31.1. Representaciones

31.2. BFS: Breadth-first Search

31.3. DFS: Depth-first Search

31.4. Dijsktra: Camino más cercano

Bibliografía

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

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