APUNTES C++

José Juan Urrutia Milán

Reseñas

- Documentación C++: https://cplusplus.com/

Siglas/Vocabulario

- **API:** Interfaz de Programación para Aplicaciones.
- **IDE:** Entorno de Desarrollo Integrado (App para programar).
- **POO:** Programación Orientada a Objetos.
- **OS:** Operation System
- **Layout:** Es la forma en la que se disponen los diferentes elementos en un marco o lámina (no es una sigla).
- **Cte/Ctes:** Abreviatura de Constante/Constantes.
- **Parámetro de tipo:** Tipo de dato en programación genérica (Se especifica dentro de <>).
- **BBDD:** Bases de Datos.
- **MVC:** Modelo Vista Controlador

Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a: Hace referencia a cualquier caracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var: Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

Índice

stol(string s)

to_string(... n)

max(int a, int b) / min(int a, int b)

sizeof(... x)

sizeof

isalphanum(char c)

isalpha(char c)

isblank(char c)

isdigit(char c)

isupper(char c) / islower(char c)

toupper(char c) / tolower(char c)

Librería cmath

pow(double base, double exp)

sqrt(double x)

abs(double x)

exp(double x)

log(double x)

log10(double x)

round(double x)

ceil(double x)

floor(double x)

Título III: cout, cin

cout

cerr

clog

cin

endl

getline()

Librería iomanip

setw(int x)

setprecision(int x)

setfill(char c)

Modificadores de cout

dec / hex / oct

showbase

uppercase

right / left

fixed / default / scientific

showpoint

boolalpha

Título IV: Condicionales

if, if else, else

switch

default

Operador ternario

Título V: string

Métodos

length()

empty()

clear()

append(string s)

at(int i)

insert(int i, string s)

find(char c)

erase(int pos, int length)

substr(int pos, int length)

Métodos constructores

string()

string(string s)

string(string s, int n)

string(int n, char c)

string(string s, int begin, int length)

Título VI: Bucles

while

do while

for

foreach

break

continue

Título VII: Funciones

return

Parámetros con valor por defecto

Parámetros constantes

Paso de punteros

Paso por valor y paso por referencia

Parámetros output

Paso de arrays

Funciones inline

Funciones modelo / blueprint

Diferentes tipos de datos

Especificar tipo de dato

Paso por referencia

Especialización de funciones

Título VII.I: Funciones lambda

Asignación a una variable

void

Lista de captura

Paso por valor y paso por referencia

Capturar todas las variables por valor

Capturar todas las variables por referencia

Título II: Modificadores de acceso

- public
- private
- protected

Título III: Campos de clase

- static

Título IV: Métodos

- this
- const
- operator

Título V: Métodos constructores

- Constructor por defecto
- Lista de inicialización

Título VI: Métodos destructores

- Cuando son llamados

Título VII: Herencia

- public
- protected
- private
- using
- final
- Polimorfismo

Capítulo I: Programación básica

Añadir librerías

Para añadir librerías, seguiremos el siguientes esquema:
#include <nombre>

Que incluiremos en la cabecera de nuestro archivo .cpp

Las librerías más usadas son:
iostream
cmath

Namespace

Nos da una solución para prevenir conflictos de nombre en proyectos grandes.
Se suele usar el **namespace** standard de C++:

using namespace std;

Podemos incluir un namespace con la siguiente estructura:
using namespace nombre;

Función main

La función main en C++ tiene la siguiente estructura:

```
int main(){
        código;
        return 0;
}
```

Comentarios

Para añadir comentarios, contamos con los míticos / y /**/, al igual que en Java.

Operadores

Iguales a los de Java.
Contamos además con "<<" , ">>" y "::".

++ / --

Sirve para incrementar una variable numérica en una unidad.

Debemos tener cuidado con el orden: al incluirlos detrás de una variable, haremos uso de la variable y luego la incrementaremos, pero si la usamos delante, la incrementaremos y luego la usaremos:

```
int a = 5;
cout << a++ << endl;          // Imprime 5
a = 5;
cout << ++a << endl;          // Imprime 6
```

Casting

Contamos con un casting explícito:

```
int a = (int) 5.5;
```

que también podemos hacer con:

```
int a = static_cast<int> 5.5;
```

Generar números aleatorios

(**#include <ctime>**)
Primero, debemos establecer una semilla:

```
srand(time(0));
```

A continuación, podemos hacer uso de la función rand() para obtener números aleatorios:

```
int n = rand();
```

Título I: Variables

Podemos declarar una variable y luego asignarle un valor o ambas a la vez, como en Java. Seguiremos el siguiente esquema:

tipodato nombre;
nombre = valor;

o

tipodato nombre = valor;

Lista de inicialización

tipodato nombre {valor};

*En este último caso no es posible realizar un casting implícito, saltará un error de compilacion.
*Si hacemos uso de la lista vacía, inicializará la variable en un valor por defecto:

int a {}; // Inicializa a en 0.

Podemos además hacer una inicialización con paréntesis:
tipodato nombre (valor);

En este caso, sí que se aplicarán los castings implícitos.

Constantes

Para declarar una constante, haremos uso de la palabra reservada **const**.
Las constantes deben inicializarse a la vez que se declaran (salvo que sean campos de clase):

const tipodato NOMBRE = valor;

*Por convenio, el nombre de las constantes se especifica en mayúsculas.

Referencias

Podemos crear variables que referencien a otras:
tipodato &nombre = variable;

*No podemos declarar una referencia y luego inicializarla:
int &referencia; // Error de compilción

Si modificamos la referencia a una variable, la variable quedará modificada y viceversa.

Cuando usamos una referencia, no incluimos &:

```
int a = 5;
int &ref = a;
ref = 6;
```

Las referencias son útiles cuando queremos pasar a una función no valores sino variables.

*Una referencia no puede cambiar la variable a la que se refiere.

const

Podemos crear referencias constantes a variables.
Al ser constante la referencia, no podremos modificar su valor.
Sin embargo, si modificamos el valor de la variable, el valor de la referencia constante también será modificado.

Para declarar una referencia constante:

const tipodato &nombre = variable;

*Sólo podremos crear referencias constantes a variables constantes.
**Los pasos por referencia constante son iguales que por valor pero nos permiten ahorrar memoria. Y además permite que se le especifiquen expresiones como parámetros a una función que recibe una referencia constante.

Posición de &

La posición de & no es relevante:
int var;
int& ref = var; **int & ref = var;** **int &ref = var;**

Tipos de datos

short, int, long, long long, size_t …
double, float, …
char
bool
string (no es primitivo, al igual que en Java)

size_t permite almacenar enteros positivos bastante grandes.

*Nota: Una variable bool se reconoce internamente como un entero, donde la variable adquiere el valor **false** si esta vale 0 o **true** en caso contrario (usualmente se asigna el valor 1).
De esta forma, el siguiente código imprimirá en pantalla un 1:

```
bool variable = true;
cout << variable;
```

*Podemos usar literales float o long, debiendo indicar el tipo de dato con un sufijo:

```
auto a = 5.6f;      // = (float a = 5.6;)
auto a = 5l;        // = (long a = 5;)
auto a = 5ll;       // = (long long a = 5;)
```

*Se pueden usar literales en notación científica en C++:
long long a = 5e10;

Librería limits

Podemos conocer el valor máximo y mínimo que puede almacenar un tipo de dato:

#include <limits>
(dentro del main:)
numeric_limits<tipodato>::min();
numeric_limits<tipodato>::max();

Nos permiten conocer los valores mínimo y máximo de cada tipo de dato.

typedef

Podemos usar un "alias" para tipos de datos demasiado largos, usando la siguiente estructura:
typedef tipodato alias;

Ejemplo:
typedef std::vector<std::pair<std::string, int>> lista_t;

A partir de ahora, en vez de escribir **std::vector<std::pair<std::string, int>>,** podremos escribir **lista_t** en su lugar.

*Por convenio, se suele especificar "_t" en los alias.

using

Además de para incluir namespaces, podemos usar la palabra **using** de una forma similar a typedef:
using alias = tipodato;
De esta forma, obtendremos la misma funcionalidad que con typedef.

Se recomienda el uso de using sobre typedef.

auto

Disponemos del tipo de dato auto, donde el compilador asigna el tipo de dato que considera oportuno a la variable (similar a las variables normales de python).

auto a = 5;

signed / unsigned

Por defecto, las variables enteras serán signed:

int a = 5; // = (signed int a = 5;)

De esta forma, la mitad de la memoria reservada será para números positivos y la otra mitad para negativos.

Si usamos la palabra unsigned, conseguiremos que nuestra variable sólo pueda almacenar números positivos, de forma que la cantidad de números positivos que podemos almacenar se dobla y la cantidad de números negativos pasa a ser 0.

unsigned int a = 5;

*Podemos usar literales unsigned:
Deberemos incluir “u” al final del literal para indicar que se trata de un literal de tipo unsigned:

auto a = 5u; // = (unsigned int a = 5;)

Además, disponemos de “ul” para unsigned long.

Ver posición en memoria

Podemos usar el operador **&** para ver la posición en memoria de una variable:

```
int n = 5;
cout << &n << endl;
```

Sistema numérico

Por defecto, C++ detectará que los literales están en decimal, pero podemos incluir en el código literales en octal, hexadecimal o binario:

0

Incluir un 0 delante de nuestro literal hará que C++ interprete que dicho literal se encuentra en octal:

int a = 010; // a = 8 = 10_8

0x

Incluir 0x delante de nuestro literal hará que C++ interprete que dicho literal se encuentra en hexadecimal:

int a = 0x10; // a = 16 = 10_{16}

0b

Incluir 0b delante de nuestro literal hará que C++ interprete (a partir de C++14) que dicho literal se encuentra en binario:

int a = 0b10; // a = 2 = 10_2

Título II: Funciones útiles

stoi(string s)

Convierte devuelve el casting a **int** de s.

stod(string s)

Convierte devuelve el casting a **double** de s.

stol(string s)

Convierte devuelve el casting a **long** de s.

to_string(... n)

Convierte devuelve el casting a **string** de n.
(**...** indica que existe una sobrecarga de funciones **to_string** para los diferentes datos de tipo primitivo)

max(int a, int b) / min(int a, int b)

Devuelve el máximo o el mínimo entre a y b.

sizeof(... x)

Devuelve (int) el tamaño en bytes que ocupa la variable x.

*Podemos indicarle directamente un tipo de dato, sin tener que pasarle una variable de dicho dato:

sizeof(double);

De esta forma, podemos calcular el numero de elementos de un array:

```
int array[] = {1, 2, 3, …};
int elementos = sizeof(array) / sizeof(array[0]);
```

sizeof

Además, existe esta palabra reservada que funciona de forma similar a la función sizeof()

isalnum(char c)

Determina si c es un carácter alfanumérico.
(Devuelve un int pero cuando sea falso, será 0 así que podemos usarlo en un if)

isalpha(char c)

Determina si c es un carácter alfabético.

isblank(char c)

Determina si c es un espacio.

isdigit(char c)

Determina si c es un dígito.

isupper(char c) / islower(char c)

Determina si c es una mayúscula / una minúscula.

toupper(char c) / tolower(char c)

Devuelve (char) el caracter c en mayúscula / minúscula.

Librería cmath

pow(double base, double exp)

Eleva base a exp.

sqrt(double x)

Devuelve la raíz cuadrada de x.

abs(double x)

Devuelve el valor absoluto de x.

exp(double x)

Devuelve (double) e^x.

log(double x)

Devuelve (double) ln(x).

log10(double x)

Devuelve (double) $\log_{10}(x)$.

round(double x)

Devuelve x redondeado.

ceil(double x)

Devuelve el siguiente número entero que sigue a x.

floor(double x)

Devuelve la parte entera de x.

Título III: cout, cin

cout

cout nos permite imprimir texto en consola.
Así quedaría el hola mundo en C++:

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

```
int main(){
	cout << “Hola mundo” << endl;
	return 0;
}
```

El operador “<<” (inserción) nos permitirá concatenar texto con variables:

cout << “Hola, me llamo “ << nombre << “ y tengo “ << edad << “ años.” << endl;

*Cuando vayamos a imprimir el resultado de evaluar una expresión, se recomienda usar paréntesis:

```
string solucion, respuesta;
cout << “Ha acertado: “ << (solucion == respuesta) << endl;
```

cerr

Imprime un error en pantalla.

clog

Imprime un log.

cin

cin nos permite leer palabras desde consola.
Así leeríamos una palabra:

```
string texto;
cin >> texto;
```

Además, podemos concatenar lecturas de palabra con el operador “>>”:

```
string a, b, c;
cin >> a >> b >> c;
```

endl

Nos permite imprimir un salto de línea en consola.

getline()

Nos permitirá leer una línea entera:

```
string linea;
getline(cin, linea);
```

*Si mezclamos **cin** con **getline**, para evitar leer un salto de línea con getline, usaremos lo siguiente:

```
string linea;
getline(cin >> ws, linea);
```

Librería iomanip

setw(int x)

Indica un tamaño mínimo a la hora de imprimir un texto en consola:

```
cout << setw(10) << "Hola";          // Imprime "      Hola"
```

setprecision(int x)

Indica el número de decimales que se mostrarán en consola.
Por defecto, se muestran 6.

setfill(char c)

Cambiar el caracter de relleno (usar con setw()). (Por defecto es ' ').

Modificadores de cout

dec / hex / oct

Además, podremos hacer uso de las palabras reservadas **dec**, **hex**, **oct** para imprimir en pantalla variables en diferentes sistemas numéricos:

```
int var = 10;
cout << dec << var << endl;          // imprime 10
cout << hex << var << endl;          // imprime a
cout << oct << var << endl;          // imprime 12 (10 = 12_8)
```

showbase

Este modificador de cout nos permitirá ver los prefijos (0, 0x, 0b, …) delante de los números que se imprimen:

```
cout << showbase << hex << var << endl;          // imprime 0xa
cout << showbase << oct << var << endl;          // imprime 012
```

noshowbase nos permite lo contrario, ocultar estos prefijos (por defecto).

uppercase

Permite que los números hexadecimales que se impriman en consola lo hagan con letras mayúsculas.

```
cout << uppercase << showbase << hex << var << endl;          // imprime 0XA
```

nouppercase nos permite lo contrario (por defecto).

right / left

Indica una justificación de texto:

```
cout << right << setw(10) << "Hola" << endl;
cout << left << setw(10) << "Hola" << endl;
```

fixed / default / scientific

default usa la notación científica o los números enteros cuando es necesario (por defecto).

fixed imprime todos los números con 6 decimales.

scientific imprime todos los números en notación científica (1 entero, 6 decimales y un exponente de 10).

showpoint

Rellena la parte de los decimales con 0s hasta el setprecision() actual.

noshowpoint consigue el efecto contrario (por defecto).

boolalpha

Para visualizar el correspondiente “true” en consola, deberemos especificar:

```
bool variables = true;
cout << boolalpha << variable;
```

*Podemos usar **noboolalpha** para el efecto contrario, para volver a mostrar las variables bool como 0 o 1 (por defecto).

Título IV: Condicionales

if, if else, else

Iguales a los de Java:

```
if(condición){
        código;
}else if(condición){
        código;
}else{
        código;
}
```

Si el bloque de un **if / if else / else** es de una única línea no será necesario indicar las llaves:

```
if(edad >= 18)
        cout << "Eres mayor" << endl;
else
        cout << "Eres menor" << endl;
```

switch

Igual al de Java:

```
switch(variable){
        case valor:
                código;
                break;

        case valor2:
                código;
                break;
}
```

*Las variables de los switches sólo podrán ser valores enteros.

default

Se incluye como último case, si ninguno de los case fue verdadero, el flujo de ejecución entra en el default:

```
switch(edad){
        …
        …
        default:
                cout << "Edad inválida" << endl;
                break;
}
```

Operador ternario

El operador ternario es una expresión que nos permite devolver un valor u otro en función de una condición, siguiendo la siguiente estructura:

condición ? verdad : falso;

Si la condición es cierta, se devolverá **verdad**, **falso** en caso contrario:

```
int edad;
```

bool mayor_edad = edad >= 18 ? true : false;

int n;
n % 2 == 0 ? cout << "Es par" << endl : cout << "Es impar" << endl;

Título V: string

Las variables string, al ser realmente un objeto, no hará falta inicializarlas, se inicializan por defecto en la cadena "".

*Podemos utilizar el tipo de dato string como si fuera un array de caracteres:

string s = "Hola";
cout << s[0] << endl; //Imprimirá H

*Podemos usar el operador "+" para concatenar strings.
Sólo podremos concatenar strings con strings.

Métodos

length()

Devuelve (int) el número de caracteres de la cadena.

empty()

Devuelve (bool) si la cadena está vacía o no.

clear()

Reemplaza la cadena por una vacía.

append(string s)

Concatena s a la cadena sobre la que se ejecuta.

at(int i)

Devuelve (char) el carácter que se encuentra en la posición i.
(El primer carácter de la cadena tiene la posición 0).

insert(int i, string s)

Introduce la cadena s en la posición i de la cadena, desplazando a la derecha de s el contenido de la cadena original que quedaba detrás de la posición i.

find(char c)

Devuelve (int) la posición de la primera aparición de c en la cadena.

erase(int pos, int length)

Elimina length caracteres a partir de la posición pos (incluido).

substr(int pos, int length)

Devuelve (string) una nueva cadena subcadena de la original de longitud length que comienza en la posición pos.
*Si se omite el parámetro pos, este valdrá 0.
*Si se omite el parámetros length, se hará una subcadena hasta el final de la cadena.

Métodos constructores

string()

Crea una cadena de caracteres vacía.

string(string s)

Crea una cadena de caracteres que contiene s.

string(string s, int n)

Crea una cadena de caracteres que contiene lo n primeros caracteres de s.

string(int n, char c)

Crea una cadena que contiene n veces el caracter c.

string(string s, int begin, int length)

Crea una cadena que contiene length caracteres, una copia de s a partir del caracter en la posición begin.

Título VI: Bucles

while

Igual al de Java:

```
while(condición){
        código;
}
```

do while

Igual al de Java:

```
do{
        código;
}while(condición);
```

for

Igual al de Java:

```
for(variable; condición; incremento){
	código;
}

for(int i = 0; i < 10; i++){
	cout << to_string(i) << endl;
}
```

foreach

Igual al de Java:

```
for(tipodato variable : tipodato+){
	código;
}
```

Donde **tipodato+** es un tipo de dato de una dimensión más que **tipodato**.

```
int array[] = {1, 2, 3, 4};
for(int a : array){
	cout << to_string(a) << endl;
}
```

*En todos los bucles se podrán omitir las llaves siempre y cuando el código a ejecutar dentro del bucle sea una línea:

```
for(int i = 0; i < 10; i++)
	cout << to_string(i) << endl;
```

break

Sale del bucle actual.

continue

Termina esta iteración del bucle.

Título VII: Funciones

La estructura para crear una función será la siguiente:

```
tipodato nombre(parámetros){
	código;
}
```

C++ nos permite declarar funciones (mediante un prototipo) y luego definirlas:

```
void saludar();

int main(){
	saludar();
	return 0;
}

void saludar(){
	cout << "Hola" << endl;
}
```

C++ Permite la sobrecarga de funciones.

return

En funciones que devuelvan datos, usaremos la palabra reservada **return**.

Parámetros con valor por defecto

Cuando determinamos los parámetros que recibe una función podemos determinar que ciertos parámetros tengan un valor por defecto.
Esto nos permite llamar a la función sin especificar dichos parámetros. En ese caso, dichos parámetros adquirirán el valor por defecto. Si se especifica el valor de los parámetros, se usará el valor especificado.
Los parámetros con valor por defecto deben especificarse después de los parámetros normales.
Para especificar un valor por defecto, incluimos una asignación a un parámetro:

```
int suma(int a, int b = 1){
	return a + b;
}

suma(2, 3);	// Devuelve 5
suma(2);	// Devuelve 3
```

Parámetros constantes

Podemos determinar que ciertos parámetros no puedan cambiarse dentro de funciones, mediante el uso de la palabra reservada **const**:

```
void funcion(const int a, int b){
	a = 5;	// Fallo de compilación al ser el parámetro a constante.
}
```

Esto es especialmente útil cuando nuestra función recibe un puntero o un valor por referencia y no queremos modificar estos valores (porque entonces se modificarán también fuera de la función), por lo que declaramos los parámetros como constantes:

void funcion(const int* a, const int& b)

Paso de punteros

Para que una función reciba un puntero, deberemos indicar el símbolo "*" en la declaración de los argumentos que esta recibe. Al llamar a la función deberemos pasarle la dirección de memoria de la variable (con el operador &).
Al modificar el valor del puntero dentro de la función, modificaremos la variable a la que este apunta.

Paso por valor y paso por referencia

Cuando pasamos un parámetro a una función se realiza el paso por valor, es decir, dentro de la función se realiza una copia de la variable que fue indicada como parámetro.
Sin embargo, si queremos trabajar dentro de la función con la misma variable que teníamos, debemos pasar esta variable por referencia, indicándolo explícitamente con el operador **&** en el nombre de la variable que recibe la función.
De esta forma, la función recibe una referencia de la variable.

```
int main(){
        int a = 5, b = 6;
        intercambiar(a, b);
        cout << to_string(a) << " " << to_string(b) << endl;
        return 0;
}
```

Podríamos pensar que la función intercambiar podría tener la siguiente cabecera:
void intercambiar(int a, int b);

Pero esa función recibirá las variables por valor y no por referencia, por lo que para esto, necesitamos usar esta nueva cabecera:
void intercambiar(int& a, int& b);

Parámetros output

El paso por referencia nos permite introducir que el output de la función se lo pasemos a la función como un parámetro:
Ejemplo:

```
void maximo(int a, int b, int& resultado){
        resultado = a > b ? a : b;
}
```

Paso de arrays

Al pasar un array de más de una dimensión como parámetro hay que indicar el tamaño de todas las dimensiones salvo de la primera.

Funciones inline

Funciones pequeñas que son llamadas con mucha frecuencia, mejora la eficiencia del programa.
Deben incluirse en los ficheros .h.
Los métodos de clase implementados directamente dentro de las clases son directamente métodos inline.

Funciones modelo / blueprint

Podemos declarar funciones que nos sirvan para cualquier tipo de dato, sin necesidad de tener que sobrecargarla para cada dato.
Para ello, usaremos el dato **T**, previamente definiendo qué es T:

```
template <typename T>

T maximo(T a, T b){
        return a > b ? a : b;
}
```

Cuando se llame a estas funciones el compilador usará este blueprint para generar la función natural correspondiente.

*Debemos incluir el template cada vez que escribamos el prototipo o definamos la función:

```
template <typename T>
T funcion(parámetros);
…
template <typename T>
T funcion(parámetros){
        código;
}
```

*Nuestras funciones pueden darnos fallos si le pasamos un tipo de dato que no soporta un operador que usamos dentro de la función.

Diferentes tipos de datos

En el ejemplo, a y b son del mismo dato, para crear una función que pueda recibir dos variable cualesquiera:

```
template <typename T, typename U>

auto maximo(T a, U b){
        return a > b ? a : b;
}
```

Especificar tipo de dato

Podemos especificar qué tipos de datos estamos enviando a una función blueprint:

```
template <typename T> funcion(T a, T b){...}

int a;
double b;

funcion<int>(a, b);
```

En el ejemplo, le estamos indicando al compilador que T = int, por lo que se realizará un casting de double a int para el valor de la variable b al llamar a la función.

*Esto también puede dar un error de compilación si intentamos un casting imposible:

```
string a;

funcion<double>(a, a);        // cannot convert string to double
```

Paso por referencia

Es posible además pasar parámetros por referencia.
Estas referencias deberán ser constantes:

```
template <typename T> const T& nombre(const T& parámetros){...}
```

Especialización de funciones

Podemos hacer que una función blueprint realice cosas diferentes cuando se le especifica un tipo de dato en concreto:

```
template <typename T> T nombre(parámetros) {...}        // Función normal
template <> tipodato nombre<tipodato2>(parámetros) {...}
```

La segunda función devolverá un dato del tipo **tipodato** y se usará dicha definición cuando a la función se le indiquen variables del tipo **tipodato2**.

Ejemplo:

```
template <typename T>
T cubo(T a){
```

```
    return a*a*a;
}

template <>
const char* cubo<const char*>(const char* a){
    return a;
}

int main(){

    cout << cubo("a") << endl;          // Imprime a
    cout << cubo(3) << endl;            // Imprime 27

    return 0;
}
```

Título VII.I: Funciones lambda

Para crear una función lambda, seguiremos el siguiente esquema:

```
[listaDeCaptura](parámetros) -> tipodato {
        código;
}
```

*Podemos no indicar el tipo de dato que devuelve esta y el compilador se encargará de asignarlo.

Una vez que definimos nuestra función lambda, podemos llamarla directamente indicando dos paréntesis al final:

```
[](){
        cout << “Hola mundo”;
}();

[](int a, int b) ->int{
        return a > b ? a : b;
}(4, 5);
```

Asignación a una variable

Una vez que tenemos definida nuestra función lambda, podemos asignarla a una variable:

```
auto func = [](){
        cout << “Hola”;
};
```

Para llamar a dicha función, usaremos la sintaxis:
func();

Si al definir la función incluimos los dos paréntesis de llamada, nuestra variable almacenará directamente el output de la función:

```
auto c = [](int a, int b) ->int{
        return a > b ? a : b;
}(4, 5);
cout << c;      // Imprime el máximo entre a y b
```

void

Para crear una función lambda void haremos lo siguiente:
[](parámetros){
        código;
}

Lista de captura

La lista de captura nos permite determinar qué variables del entorno exterior de la función pueden ser accesibles dentro de la función:

El siguiente código nos dará un error de compilación ya que la variable a no es accesible desde la función
```
int a = 5;
cout << [](){return a;}();
```

Funciona perfectamente
```
int a = 5;
cout << [a](){return a;}();
```

Paso por valor y paso por referencia

Debemos tener en cuenta que la lista de captura funciona de forma similar a la lista de parámetros, creando una copia de los valores que se le indican por lo que el siguiente código:

```
int a = 5;
auto func = [a](){cout << a;};

a++;
func();
```

Imprimirá 5.

Por tanto, debemos tener en cuenta cuando indicamos valores por valor y cuando por referencia en las listas de captura.

Capturar todas las variables por valor

Podemos capturar todas las variables por valor en una función lambda con la siguiente sintaxis:

[=](parámetros){
    código;
}

Capturar todas las variables por referencia

Podremos también capturar todas las variables por referencia en una función lambda con la siguiente sintaxis:

[&](parámetros){
    código;
}

Título VIII: Arrays

Podemos declarar un array y luego inicializarlo o ambas a la vez, la sintaxis cambiará:

tipodato nombre[tamaño];

tipodato nombre[tamaño] = {valor1, valor2, …}

*Cuando declaramos e inicializamos a la vez un array, si no indicamos el tamaño de este, el compilador entiende que tendrá un tamaño de tantos elementos como hayamos introducido en él:

int array[] = {1, 2, 3}; // Crea un array de 3 casillas.

*Si indicamos un tamaño e indicamos menos elementos de los indicados crearemos un array del tamaño especificado con los primeros elementos especificados y el resto inicializados en un valor por defecto:

int array[2] = {1}; // {1, 0}

Para acceder a una posición en un array, usaremos el siguiente esquema:
nombre[posicion]

Teniendo en cuenta que el primer valor de un array es el 0:

```
int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for(int i = 0; i < 10; i++)
        cout << to_string(array[i]) << endl;
```

También podemos inicializar arrays de la siguiente forma:
int array[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

const

Podemos declarar arrays constantes.
Estos deberán inicializarse en su declaración:
const int array[] = {2, 5, 7};

size(... array)

Nos permite conocer la cantidad de casillas de un array.

sizeof

Podemos conocer el tamaño de un array usando sizeof:

```
int array[] = {1, 2, 3, …};
int elementos = sizeof(array) / sizeof(array[0]);
```

string

El tipo de dato string no deja de ser un array de char así que podremos hacer cosas como:

```
string s = “Hola”;
cout << s[0] << endl;          // Imprime H

char array[] {“Hola”};
```

strlen

(#include <cstring>)
Devuelve la longitud de un array de caracteres.

Array a función

Cuando pasamos un array por una función, verdaderamente se enviará un puntero, por lo que dentro de la función no podremos usar funciones como sizeof() sobre el array, ya que estaremos trabajando sobre un puntero.

```
int array[tamanio] = {...};
funcion(array, tamanio);
…
void funcion(int array[], int tamanio){
        …;
}
```

fill(begin, end, value)

Nos permite rellenar un array:

int array[100];
fill(array, array + 100, 0);

Inicializar arrays

También podemos inicializar todas las casillas de un array en un valor "por defecto":
int array[100] {};
Crea un array de enteros inicializado en 0.

Operador de dirección de memoria

El nombre del array en sí ya indica la dirección de memoria en la que se encuentra alojado el array, por lo que no usaremos el operador & para referirnos a él:

int array[] = {1, 2, 3};
int *pArray = array;

Título IX: Punteros

Son variables que guardan una dirección de memoria de otra variable.
El tipo del puntero indicará que dicho puntero sólo puede almacenar una dirección de memoria de ese tipo de variable.

Se usa la siguiente estructura para crear un puntero:

tipodato variable;
tipodato *nombrePuntero = &variable;

Ejemplo:
string nombre = "Juan";
string *pNombre = &nombre;

A la hora de usar el puntero, si usamos **pNombre** será lo mismo que usar **&nombre**.
Si queremos usar el puntero, deberemos especificar ***pNombre**.

Usar el puntero de la forma ***pNombre** será equivalente a usar la variable **nombre**.

Para reasignar un puntero, usaremos la siguiente sintaxis:

string nombre2 = "Pepe";
pNombre = &nombre2;

*Por convenio, el nombre de los punteros suele ser p + nombre de la variable.
*El asterisco del puntero puede ir en el lugar deseado:
string* pNombre; string * pNombre; string *pNombre;
*Todos los punteros son del mismo tamaño (independientemente del tipo de puntero).

*Los arrays son punteros constantes.

++

Para usar el operador ++ sobre un puntero, deberemos incluirlo antes, no después:

```
int a = 0;
int *b = &a;
++*b;
```

Punteros como arrays

Podemos inicializar punteros como si fueran un array:

tipodato *nombre[tamanio];

Para acceder a sus posiciones:
***(nombre + posicion);**

```
char *puntero {"Hola"};
cout << puntero << endl;          // Imprime Hola
cout << *puntero << endl;         // Imprime H
cout << *(puntero + 1) << endl;   // Imprime o
```

const

Podemos además crear punteros constantes:
const tipodato *nombre;

Si tratamos de modificar el puntero, obtendremos un error de compilación.

Punteros nulos, nullptr

Punteros que no apuntan a nada.
Es una buena práctica inicializar un puntero en nulo.
Se usa el literal **nullptr.**

Ejemplo:

```
int *puntero = nullptr;
int a = 5;
```

puntero = &a;

cout << puntero != nullptr ? “Puntero asignado” : “Puntero sin asignar”;

Comprobar con if

Las dos formas siguientes son equivalentes:

if(puntero == nullptr)
if(puntero)

Punteros a funciones

Podemos definir punteros a funciones de la siguiente forma:
tipoDevuelta (*nombre)(tipoParámetros);

Ejemplo:

bool esMayor(int a, int b){...}

bool (*comparar)(int, int) = esMayor;

Uso:
(*esMayor)(3, 4);

Memoria dinámica

Podemos crear espacios en memoria (heap) y borrarlos dinámicamente:

```
int *puntero = new int;          // Equivalente a:
*puntero = 21;                   // int *puntero {new int {21}};

cout << “Direccion: “ << puntero << endl;
cout << “Valor: “ << *puntero << endl;

delete puntero;
```

Con arrays, la nomenclatura es la siguiente:

```
char *puntero = nullptr;
puntero = new char[tamanio];
…
delete[] puntero;
```

Puede ser una buena técnica el usar la memoria dinámica cuando tenemos que crear arrays cuyo tamaño depende del input del usuario.

*Debemos tener cuidado de no dejar espacios en el heap ocupados:

int *puntero {new int{67}}; //Ocupa espacio en el heap

int numero {68};
puntero = №

Ahora hay un espacio en el heap al que nuestro programa no tiene acceso, memory leak. Por tanto, antes de asignar un puntero a una nueva dirección de memoria, debemos eliminarlo:
delete puntero; (antes de puntero = &numero)

nothrow

El uso de new puede lanzar una excepción si no encuentra espacio en la memoria, por ejemplo, cuando queremos buscar un espacio para un array muy grande.
Podemos hacer que no se lancen excepciones:

int *puntero {new(nothrow) int};

Si no se consigue asignar el puntero, adquirirá el valor de nullptr.

Arrays dinámicos

Arrays situados en el heap.
Para crear uno:
tipodato *nombre = new tipodato[tamanio];

Podremos acceder a sus valores de dos formas:
int *puntero = new int[100]{};

La sintaxis de array:
int elemento = puntero[0];

La sintaxis de puntero:
int elemento = *puntero;
int elemento2 = *(puntero + 1);

delete

delete[] puntero;
puntero = nullptr;

Título X: Structs

Un struct es un grupo de variables relacionadas sobre un nuevo "tipo de dato".
La estructura de un struct es la siguiente:

```
struct nombre{
        tipodato campo1;
        tipodato campo2;
        …
};
```

A partir de un struct podemos crear "objetos" pertenecientes a dicho struct.
Cada objeto tendrá tantos campos como se indicaron en el struct.

Para crear un objeto de un struct:

```
nombreStruct nombre;
nombre.campo1 = valor1;
nombre.campo2 = valor2;
…
```

ó

```
nombreStruct nombre = {valor1, valor2, …};
```

Ejemplo:

```
struct Persona{
        string nombre;
        int edad;
}

int main(){
        Persona persona1;
        persona1.nombre = "Juan";
        persona2.edad = 15;

        Persona persona2 = {"Maria", 16};
}
```

Los campos de clase de un struct son públicos por defecto.

Título XI: Enumerados

Los tipos enumerados es un tipo de dato creado por el programador que puede adoptar valores finitos. Internamente, los enumerados son un tipo de dato entero.
Para crear un tipo enumerado, seguiremos el siguiente esquema:

enum nombre {valor1, valor2, …};

Posteriormente, para crear una variable de este tipo enumerado lo haremos de la forma canónica:

nombreStruct nombre = nombreStruct::valori;

Ejemplo:

```
enum Estaciones {primavera, verano, otonio, invierno};
Estaciones estacion = Estaciones::primavera;
```

ó

```
Estaciones estacion = primavera;
```

*Cada valor tiene asociado por defecto un valor numérico tal y como se especifica a continuación. Pese a ello, podemos cambiar los valores manualmente.

```
enum Estaciones {primavera = 0, verano = 1, otonio = 2, invierno = 3}
Estaciones estacion = primavera;
if(estacion == 0)
        cout << “Estamos en primavera” << endl;
```

Capítulo II: POO

Título I: Clases

Las clases en C++ siguen la siguiente estructura:

class nombre{
código;
};

*Una clase o struct puede tener varias definiciones pero tienen que estar en diferentes unidades de traducción.

Objetos

Para crear un objeto (una "variable" del tipo de dato de una clase) a partir de una clase, haremos lo siguiente:

nombreClase nombre;

Ejemplo:
class Persona{...};

Persona p;

Objetos mediante punteros en el heap

Podemos crear objetos mediante punteros que se almacenarán en el heap:

Persona* p = new Persona;

Para usar los métodos de la clase persona, podemos usar dos notaciones distintas:
(*p).correr();
p->correr();

Que no se nos olvide borrar la información del heap al final:
delete p; // Llama al destructor de Persona.

Título II: Modificadores de acceso

En C++ cada modificador de acceso afecta a todas las líneas siguientes hasta encontrar un nuevo modificador de acceso:

class nombre{
modificador1:

```
        // Región afectada por el modificador1

    modificador2:
        // Región afectada por el modificador2
};
```

public

Los campos y métodos afectados por este modificador son accesibles tanto fuera como dentro de la clase.

private

Los campos y métodos afectados por este modificador son accesibles sólo dentro de la clase.

protected

Los campos y métodos afectados por este modificador son accesibles sólo dentro de la clase y por parte de alguna clase que herede de esta.

*El modificador de acceso por defecto es private.

Título III: Campos de clase

Los campos de clase son variables que se almacenan dentro de objetos.
Para declarar un campo de clase:

```
class nombre{
    private:
        tipodato nombreDato;
};
```

*El que los campos de clase estén afectados por un modificador de acceso private es una buena técnica de programación.
*Los campos de clase no pueden ser referencias (pueden ser punteros).

static

Cuando creamos un campo de clase static, sólo podremos asignarle un valor si se trata de un campo entero. Si se trata de un campo no entero, deberemos asignarlo fuera de la clase:

```
class nombre{
    static const entero nombre_var = valor;
};

class nombre{
    static const no_entero nombre_var;
```

```
};

const no_entero nombre::nombre_var = valor;
```

Además, los campos estáticos no constantes deberemos inicializarlos de la forma siguiente:

```
class nombre{
        static int nombre_var;
};

int nombre::nombre_var = valor;
```

Título IV: Métodos

Los métodos son funciones que pertenecen a una clase.
Para declara un método:

```
class nombre{
        tipodato nombreMetodo(parámetros){
                código;
        }
};
```

this

En los métodos que reciban argumentos con el mismo nombre que un campo de clase, usaremos **this->** delante del nombre del campo de clase para diferenciarlo de la variable de los parámetros:

```
class Persona{
        private:
                string nombre;
                int edad;

        public:
                Persona(string nombre, int edad){
                        this->nombre = nombre;
                        this->edad = edad;
                }
};
```

Cada método contiene un puntero por defecto (this) que contiene la dirección de memoria del objeto actual sobre el que se ejecuta el método.

const

Podemos crear métodos const.
Dichos métodos no podrán modificar los campos de clase del objeto y darán un error de compilación en caso de que lo intentemos:

```
class Coordenada{
	private:
		int x;
	public:
		int getCoordenada() const{  // No puede modificar x
			return x;
		}

		int getCoordenada2() const{
			x++;		// Error de compilación
			return x;
		}

		int getCoordenada3() {
			x++;		// legal, el método no es const
			return x;
		}
};
```

operator

Podemos cambiar el comportamiento por defecto de un operador de c++ con un objeto de la siguiente forma:

```
class Complex{
	public:
		int a, b;

	Complex operator + (Complex){
		Complex c;
		c.a = x.a + a;
		c.b = x.b + b;
		return c;
	}
};
```

Además también podemos cambiar el operador "<<" para el cout (fuera de la clase):

```
ostream &operator<<(ostream &out, const Complex &c)
```

```
{
   out << c.toString();
   return out;
}
```

*Nota: el método toString() debe ser const.

Título V: Métodos constructores

La sintaxis de un método constructor es la siguiente:

```
class nombreClase{
        public:
                nombreClase(parámetros){

                }
};
```

Cuando se crea un objeto de la siguiente forma:
Persona p;

Realmente se está haciendo uso del método constructor sin argumentos.

*Los métodos constructores siempre serán métodos públicos.
*Si queremos crear un array de objetos de nuestra clase, será **obligatorio** que nuestra clase tenga un método constructor que no reciba parámetros.

*Por esta razón y por ser una buena técnica de programación, será recomendable que toda clase que creamos tenga un método constructor sin parámetros.
*Además, también será una buena técnica que todas las clases tengan un método **to_string()**, un método que devuelve en un string toda la información de los campos del objeto.

C++ Permite la sobrecarga de constructores.
Al crear un objeto, el compilador decidirá qué método constructor se debe usar en cada caso.

Constructor por defecto

Todas las clases tienen por defecto un método constructor por defecto (en caso de que no se haya especificado un método constructor). Este inicializa a todos los campos en un valor basura.
Si queremos añadir el constructor por defecto a una clase:

```
class Clase{
        public:
                Clase() = default;
```

}

Lista de inicialización

En C++, los métodos constructores cuentan con una lista en la que podremos inicializar los campos de clase de una nueva forma:

Forma antigua:

```
class Persona{
        private:
                string nombre;
                int edad;

        public:
                Persona(string n, int e){
                        nombre = n;
                        edad = e;
                }
};
```

Lista de inicialización:

```
class Persona{
        private:
                string nombre;
                int edad;

        public:
                Persona(string n, int e) :nombre(n), edad(e) {}
};
```

En ambos casos el código:
Persona p("Juan", 18);
tendrá el mismo efecto.

*Si algún campo de clase es una constante, deberemos inicializarla en la lista de inicialización.

Título VI: Métodos destructores

Métodos que son llamados cuando un objeto muere.
Los destructores son útiles cuando un campo de clase es un puntero y puede haber almacenado información del heap.
Esta información deberemos borrarla cuando terminemos de usar el objeto por lo que es buena idea hacer que el destructor la borre.

La sintaxis de un destructor es la siguiente:

```
class nombre{
        ~nombre(){
                código;
        }
}
```

Ejemplo:

```
class Dog{
        private:
                int *edad;

        public:
                Dog(){
                        edad = new int {1};   // Ocupa memoria en el heap.
                }

                ~Dog(){
                        delete edad;          // Libera la memoria del heap.
                }
};
```

Cuando son llamados

Los métodos destructores son llamados cuando:

- Cuando un objeto se pasa como valor a una función.
- Cuando un objeto local se devuelve de una función (en algunos compiladores).
- Cuando un objeto local del stak muere.
- Cuando un objeto del heap es liberado con delete.

Título VII: Herencia

Las clases pueden heredar métodos y campos de clase de otras clases.
Cuando una clase hereda de otra, automáticamente se implementará en esta los métodos y campos de clase de la superclase que no sean privados.

Para hacer que una clase herede de otra, seguiremos la siguiente estructura:

```
class nombreClase : modificadorDeAcceso nombreSuperClase{
        …
};
```

(En este título nos referimos a campos de clase y métodos como “estructuras”)
Las estructuras private no pueden ser accesibles desde la clase hijo.

public

Si especificamos **public** como modificador de acceso, las estructuras públicas de la superclase seguirán siendo públicas en la clase hija. Las protected seguirán siendo protected y las private seguirán siendo private.

protected

Si especificamos **protected** como modificador de acceso, las estructuras públicas y protected de la superclase serán protected en la clase hija y las private seguirán siendo private.

private

Si especificamos **private** como modificar de acceso, todas las estructuras de la superclase serán private en la clase hija.

using

Es posible heredar constructores directamente de una superclase:

```
class Persona{
        public:
                Persona(){...}
};

class Ingeniero : public Persona{
        public:
                using Persona::Persona;      // Heredamos el constructor
};
```

final

Los métodos a los que se les especifica final no podrán ser sobreescritos en clases hijo. Dichas clases tendrán como definición del método aquella en la que se especificó final:

```
tipodato nombre(parámetros) final {
        código;
}
```

Además, podemos crear clases que hereden de forma final de otras, de forma que ninguna clase pueda heredar de esta:

```
class Ingeniero final : public Persona{...};
```

Ninguna clase puede heredar de Ingeniero.

Polimorfismo

Podemos crear punteros de la clase padre que haga referencia a objetos de la clase hijo:

Persona * persona = new Ingeniero;