APUNTES DE C++

https://en.cppreference.com/w/cpp/compiler_support

https://winlibs.com/

https://en.cppreference.com/w/

DECLARACIÓN DE VARIABLES

```
int a = 5;

int a{}; // a lo inicializa a cero pero es C++11

int a {5}; // lo inicializa a 5. es mejor llaves por que int a {2.4}; no dejaría. Solo sería válido con:
    int a {static_cast<int>(2.4)}

int array[5] = {0} //lo inicializa a cero todos los elementos del array.

int a (5); //asignación por función pero no es una función y lo inicializa a 5. C++98

----------

int a = 2.5 //seria truncado a 2.

int a {2.5} //daría error por lo que siempre es más seguro. C++11

----------

int a = 2'000'000; //esto es valido en C++11 y se usa para separar y hacer más facil lectura 2 millones.
```

PRECISIÓN DE VARIABLES REALES

```
float a = 1.123456789f; //daría en print = 1.234568 redondeando al 7º digito de precisión

double a = 1.12345678901234567; //en print = 1,234567890123457 (15 digitos precisión

long double a = 1.23456789….890L ; //15-16 de precisión pero más compatible portable.

----------
```

Los tres si se imprimen darían 6 de precisión por que es la precisión predefinida de std::out, por lo que para que imprimiera todo habría que poner:

```
std::cout « std::setprecision(20); //control de la precisión a partir de esta linea.
```

CARACTERES

```
char a { 'a' }; //asigna a como character de la variable a. C++11

----------

char a = 65;
```

```
std::cout « a; //imprime A y no 65
std::cout « static_cast<int>(a) « std::endl; //imprime 65. lo convierte a su valor de int.
```

MANIPULACION DE CARACTERES

Se usa la librería <cctype> https://en.cppreference.com/w/cpp/header/cctype

```
std::isalnum('e') //es alphanumerico o no. Suelta un int. != 0 si es. == 0 si no es. Usa <iostream>
std::isalpha(variable char) //es alfabética la variable. != 0 si lo es. == 0 si no lo es. <iostream>
std::isblank(str[i]) //Muestra si es un espacio (vacio)
std::islower(variable) //si es minúscula   std::toupper(var) //Lo vuelve a mayúsculas
std::isupper(variable) //si es mayúscula.  std::tolower(var) //Lo vuelve a minúscula.
std::isdigit(variable char) //si es dígito.
```

TIPO AUTO. SOLO EN C++11!!!

Deducirá el typo solo con la declaración:

```
auto var1 {12}; //int 4 bytes pero no solo por el auto es C++11 sino tambien por la init por llaves {}
auto var1 {12.0}; //double 8 bytes
auto var1 {12.0f}; //float 4 bytes
auto var1 {12.0l}; //long double 16 bytes
auto var1 { 'e' }; //char 1 byte
----------------------
auto var1 = 123u; //unsigned 4 bytes
var 1 = - 3; //ERROR! compilará pero pondrá un valor basura ya que es un unsigned int.
-------------------------
```

Puede ser aplicado a funciones:

```
auto funcion(int num){ //será tratado como un entero.
    return 30;}
```

TIPO BOOL

En C++ no hace falta meter librería stdbool.h como en C. Ya viene de serie.

```
bool var1 = true; // o false
std::cout « var1; // imprimirá 0 (ó 1)
std::cout « std::boolalpha;
std::cout « var1; //imprimirá false (o true) gracias al std::boolalpha.
```

ENUMERACIONES (enum)

enum class Mes { Ene, Feb, Mar, …, Nov, Dic}; //lo de dentro de {} son Enumeradores.

Mes variableMes {Mes::Feb}; //se puede crear variables de ese tipo enumerado.

std::cout « "el mes es: " « static_cast<int>(Mes) « std::endl; // 1. Cada Enum es un int por debajo. Empieza en el 0. Pero se puede modificar:

std::cout « Mes; //ERROR!!! no compilará ya que solo es una etiqueta el Feb.

enum class Mes { Ene = 1, Feb, Mar, Abr = -20, May …}; //Feb será 2. May = -19

enum class (Mes {Ene=1, Enero=1, Feb=2, Febrero=Feb…}; //Ene = Enero = 1 y ok.

El class (C++11) añadido es seguridad. Se puede hacer enum Mes{}, e implicitamente sería convertido a un entero si decimos:

enum Mes {Ene, Feb, Mar…}; //esto si es C++98

Mes mi_mes = Ene; //ya no hace falta poner el Mes::Ene para asignarlo.

std::cout « mi_mes; //aquí si funcionará, pero imprimerá 0, ya que es el indice cero. Esto puede parecer bueno, pero no tiene sentido por que podría comparar variables de clases mes > dia, al ser convertidos a int

---

sizeof(Mes); //será respresentado por un int asi que 4bytes. Es independiente del número de elementos. si se sale del rango de int entonces crasheará.

Esto es así por que cada uno de los elementos de un enum es una asociación simbólica, en donde Ene se sustituirá por 1 (en el último ejemplo), Feb por 2, etc.… en tiempo de compilación. Luego cuando ya se declara una variable como Mes mes_actual = Mes::Feb; ya sí ocupa memoria de 4 bytes cada una de las variables.

---

enum class mes {Ene = 1, Feb = 3, Mar = 5};

main(){ mes mi_mes = mes::Feb;}

mi_mes = 4; //ERROR!!! por que tiene que ser de tipo mes::

mi_mes = static_cast<mes>(4); //sí se puede pero muy peligroso

mi_mes = static_cast<mes>(5); // mi_mes = mes::Mar.

Para que se le asigne otro tipo de variables a un enum (admite solo enteros short, o long long o char, o unsigned long … No double. Se hace así:

enum class Month : unsigned char {Jan = 0, Feb, Mar….}; //va de 0 hasta 255 del ascii Jan = 0. Sería el carácter nulo \0 si se hubiera declarado sin la palabra "class"

enum class Month : std::string {}; //ERROR. no acepta std::string el enum.

Cuando definimos una variable Month variable = Month::Jan; podemos quitar el Month:: así:

enum class Month {ene, feb, mar….};

using enum Month; //como el using namespace std, para quitar el std:: SOLO EN C++20

Month mi_mes = ene; //a partir de ese using enum Month se puede usar sin el Month::

if (mi_mes != ene) {…}; //ok

PERO si tengo dos enums compartiendo una misma etiqueta:

enum class Color1 {Amarillo, Rojo};

enum class Color2 {Verde, Rojo};

using enum Color1;

using enum Color2; //habría conflicto por que Rojo es una etiqueta que pertenece a los dos. SI NO ESTUVIERA Rojo en los dos, entonces no habría problema.

MAXIMO Y MINIMO

#include <limits>

std::numeric_limits<type>::min(); / / max() //nos da el valor minimo o maximo de ese type (int, double, etc.)

std::numeric_limits<double>::lowest() //da el numero negativo. min() da el minimo positivo, pero esto es solo para los números reales (con decimal)

CONVERSIONES A OTROS TIPOS

IMPLICITAS: double x, y = 34.2; int sum = x + y;

double x = 12.3; int y = 20; x+ y //será double

EXPLICITAS static_cast<type>(variable) //lo conviertes al tipo que quieres.

STATIC

```
void contador (){
    static int contador = 0;
    contador++;
     std::cout « contador
}
contador();
contador(); //dará 1 y 2 la siguiente vez. Solo se inicializa una vez.
```

---

como variable global, Solo es visible en el archivo fuente donde esté. Si está definida en main.c solo se verá ahí y aunque haga un extern int variable_estatica en un fichero2.c no se verá en ese fichero2.

CONSTANTES

const int age = 34; //ese valor luego no se puede reasignar. Asgura frente a accidentes de cambio.

constexpr double pi = 3.14; //mismo que const pero ejecuta en tiempo de compilación y no ejecución como const. Lo cual hace que sea más rapido. Const ejecuta en ejecución y por lo tanto pierde el tiempo. Si los cálculos matematicos no van a cambiar

constexpr double circunferencia)(double radio) {return 2 * pi * radio;} //función calculada en tiempo de compilación.

NO SE PUEDE METER una variable no constexpr en una que si.

int a = 2; constexpr int funcion(a); //ERROR. no se conoce a en tiempo de compilación

---

constinit int global_variable = 100;

(en main()): global_variable = 40; //correcto. Inicializa en tiempo de compilación pero puede ser cambiada en tiempo de ejecución cosa que constexpr no puede.

---

consteval int cuadrado(int x) {return x * x;} int main(){constexpr int result = cuadrado(5); } //OK porque consteval hace que la función solo sea accesible en tiempo de compilación. NO de ejecución

por lo que hay que pasarle variables en tiempo de compilación (constexpr):

constexpr hace que PUEDA ser en tiempo de compilación pero no tiene por que ser así. consteval garantiza que lo sea en tiempo de compilación. Si con consteval no se puede evaluar en compilación se obtiene un error de compilación.

LAS VARIABLES const SON EVALUADAS EN TIEMPO DE COMPILACIÓN (si no son asignadas como otras variables:

int a = 32;

const int x = a; //aquí ya no es en tiempo de compilación sino de ejecución ya que la variable a es en tiempo de ejecución… y se ha de ejecutar para ser asignada a la x.

---

const int* p; //Puntero a constante (entera). puede almacenar otra direccion pero no modificar la variable a quien apunta. int no tiene por que ser const. int a; p = &a // es correcto. Pero

const int a; int *p = &a //da error por que podriamos modificar el valor de const int a.

int* const p; //Puntero constante de int. No puede apuntar otro lado pero si modificar el valor de la variable.

const int* const p; //puntero constante a constante (de int) no puede apuntar a otra dirección ni puede modificar el valor.

REFERENCIAS

Es hacer llamar a una variable para usarse como la variable a la que se refiere. Un alias. En muchos casos funcionan como punteros.. PERO NO LO SON. y lo bueno es que no se genera nueva memoria para ser creados.

int variable {45};

int& r_variable = variable // o en vez de = variable, {variable}. r_ es solo un nombre. No hace falta.

r_variable = 32; //variable también será 32.

la referencia se ha de declarar. No sepuede dejar aquí:

int& r_var; // ERROR al compilar! tiene que hacer int& r_var {variable}

---

int var1 = 32;

int var2 = 45;

int &r_var1 = var1; //el & puede estar al lado de la variable como los * de los punteros.

r_var1 = var2; //solo hace que var1 tome el valor 45. NO referencia a una nueva variable var2

---

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

int (&b)[3] = array; //hay que poner el tamaño entre los corchetes obligatoriamente. Y los paréntesis también. &b[3] no es valido. ESTO TAMBIÉN PARA PARAMETRO DE FUNCIÓN.

b[2] = 8; // modificaría el a[2] a valor 8.

CONSTANTES EN REFERENCIAS.

int var = 4;

const int& r_var = var;

var = 3; // se puede y r_var será 3 ahora.

r_var++; //ERROR!! no se puede por que es constante.

pero…

const int a = 32;

int& b = a; b++; //ERROR!! violaría la constante de a. tiene que ser const int& b = a;

Auto y referencias:

int var1 = 5;

int& var1_ref = var1;

auto var2_ref = var1_ref; //será una copia. no deduce que será una referencia. Para hacerlo referencia:

auto& var2_ref = var1_ref; //este sí ya es una referencia.

¿PARA QUE SIRVE? – MODIFICAR DATA POR RANGE LOOP:

1. int array[] {1,2,3,4};

for (auto valor : array){ valor = valor*10}; //no va a modificarlo. por que valor es COPIA en loop.

for (auto& valor : array{valor = valor*10}; // SI va a modificarlo.

2. PASAR VALOR A FUNCION Y NO HACER COPIA PERDIENDO MEMORIA.

```
void funcion(int& num){num++;}
int a = 5;
funcion(a); //a pasa a ser 6;
```

ALIAS DE TIPO

```
using nombre = unsigned long long int; //como el using enum Months
nombre gran_numero {18'454'543'578'643'213ull};
nombre gran_numero2 {21'545'484'481'161'620ull};
```

Como en C:

```
typedef unsigned long long int nombre; //lo mismo que con using.
```

Pero solo podemos hacer typedef o using con tipos reconocidos (int, double, etc.) no con palabras

```
typedef patata tuberculo; //error!!! eso se hace con #define patata turberculo → tuberculo = patata
```

```
#define patata int;
patata variable = 4; //es correcto cambia patata por int.

#define pi 3,141592
double area = pi * 2 * radio; //pi = 3,1415
```

SALIDA DE DATOS

```
std::cout « "hola mundo" « std::endl; //standard
std::cerr « "error" « std::endl; //errores
std::clog « "cuidado" « std::endl; //logs
std::cin » variable; //introducción de datos
std::cin.getline(variable, longitud_chars); //para pasarle nombres con espacios.
```

Para redirigir los errores hacia un archivo de texto donde verlos en C++ los errores van asociados como en C a la salida 2 (stderro(2)), por lo tanto: se puede hacer en el bash de linux

```
./mi_programa  2> errores.log //imprime los errores en errores.log
```

Para combinar stdout(1) y stderr(2) en el mismo archivo escrito en el bash de linux:

```
./mi_programa > salida.log 2 >&1
```

FORMATOS DE SALIDA

Hay que incluir dos includes <ios> o <iomanip>

Info: https://en.cppreference.com/w/cpp/io/manip

std::endl; //imprime final de linea #include <ostream>

std::cout « "hola mundo\n"; //imprime final de linea también (mejor!!!)

---

std::setw() (set width) manipula el formato de salida tabulado al último carácter:

std::cout « std::setw(10) « "Lastname" « std::setw(10) « "Firstname" « std::setw(5) « "Age";

std::cout « std::setw(10) « "Daniel" « std::setw(10) « "Gray" « std::setw(5) « "25";

(incluir \n): Solucion:

Lastname Firstname Age //antes de Lastname(8chars) hay 2 chars, de Firs.me(9) hay 1

    Daniel   Gray   25 //antes de daniel(6chars) hay 4 chars, Gray(4) hay 6 antes…

Se puede justificar a la Derecha o Izquierda con:

std::cout « std::right; (o std::left)

std::setfil('-'); //imprimirá el carácter – como espacios: #include <iomanip>

std::cout « std::left;

std::cout « std::setfill('*');

(mismo Lastname con Daniel Grey que antes… Solución:

Lasname**********Firstname**********Age** //despues hasta completar 20/20/5 espacios.

Daniel***************Gray***************25***

std::internal; justificará el signo + o – hacia la izquierda - 123 en vez de -123.

se incluye con #include <iomanip> el setw. ,,,,left/right con <ios>

---

std::showpos muestra el + para numeros positivos, std::noshowpos los oculta <ios>

#include <ios>

std::dec (muestra valores en Decimal), std::oct (en octal), std::hex (en hexadecimal)

std::showbase (muestra 0 antes de octal, 0x en hex) // std::noshowbase (lo desactiva)

---

std::uppercase (muestra el string en mayúsculas) // std::nouppercase(lo desactiva)
<ios>

---

double c{1.34e-10};

std::cout « c « std::endl; //imprime 1.34E-10

std::cout « std::fixed; //al hacer el cout lo imprimirá como 0.000000. <ios>

---

std::scientific Muestra el valor en conotación cientifica (num x e^n) <ios>

para volver el float a su standard una vez activado el scientific hay que poner

std:cout.unsetf(std::ios::scientific | std::ios::fixed); //es un hack para quitarlo

---

#include <iomanip>

std::setprecision(numero); muestra la precisión de los numeros mostrados no solo como fracción sino también la parte antes del decimal. X defecto la precisiónd d cout es 6.

---

std::showpoint; mostrará el decimal aunque el valor sea cero. 12.0 mostrará 12.0000 y no 12. Muestra tantos ceros como la precisión tenga (por defecto 6 en cout). Si es un int no mostrará ningun decimal. int a = 32 con showpoint será siempre 32. #include <ios>

---

std::flush Manda directamente a la terminal el mensaje en vez de al buffer de salida.

std::cout « "hello world" « std::endl « std::flush; //no vemos cambio pero lo hace directo

es decir la salida lo hace en tiempo real. \n (secuencias de escape) no lo manda directo.

FUNCIONES MATEMATICAS

#include <cmath> https://en.cppreference.com/w/cpp/header/cmath

std::floor(7.7); // 7 std::ceil(7.7); //8 Redondea a la baja o al alza el valor

std::abs(-43); // 43 Da el valor absoluto.

std::exp(10); // e=2,71828^10 exponencial

std::pow(3, 4); // $2^4 = 16$ potencias

std::log10(1000); // en base 10^x=1000 logaritmo

std::log(4.5); // e^x=4.5 x defecto e

std::sqrt(81); //9 raiz cuadrada.

std::round (3.23); // 3 (3.5 = 4) Redondea como siempre.

std::sin(0.4) //cos, tan, acos, atan – seno, coseno, tangente, arcotangente.

std::rand(); //Genera un número aleatorio entre 0 y RAND_MAX. No hace falta include.

std::rand() % n; //generará numeros aleatorios entre 0 y n-1 pero siempre mismos

std::srand(std::time(0)); //sí hará distinto. time(0) da fecha. #include<ctime>

CONDICIONALES

if (var1 == var2) {} else {}; //si pasa esto entonces lo haces y si no (else) pues lo otro.

tambien se puede meter el else if para seguir con condicionales. con múltiples else if, se ejecuta uno de ellos pero no el resto.

---

switch (tool) { //el condicional tool puede ser solo int o enum (int, long, char, unsigned...NO un string)

case lapiz : {lo que tenga que hacer;}

break; //hay que poner el break o si no saltaría al otro case

case boli : {lo otro que hacer;}

break;

default : {lo final que hacer;}} //cuando no es ninguno de los casos posibles.

TERNARIOS: result = (condicion) ? opcion1 : opcion2 ;

si se cumple la condicion, entonces el resultado = opcion1. Si no, result = opcion2

Opción1 y opción2 deben ser del mismo tipo o convertibles al mismo tipo. Si no ERROR.

int speed {condicion ? 300 : 100}; //inicializará la variable speed a un valor u otro dependiendo de la condición que se de.

max = (a > b)? a: “hello”; //ERROR no son compatilbles las dos opciones.

REPETICION

FOR: for (int i {}; i < n; i++){lo que sea;} // i toma valor 0

size_t es un alias para algunos tipos de unsigned int. Normalmente tiene 8 bytes.

for (size_t i = 100; i > 0; i--){}

///IMPORTANTE: el valor de i solo va a estar DENTRO del for. No se puede acceder fuera. Para ello se ha de declarar fuera:

size_t j{}; //toma valor 0

for (j; j < 10; ++j) //aquí al salir del for, j tendría valor de 10. tambien ok : for (; j < 10; ++j)

PUEDE SER UN LOOP FOR sin interador de suma sino una lista:

FOR NO TRADICIONAL. LOOP BASADO EN EL RANGO:

*for (double multiplicador{4}; auto i : {2, 4, 6, 89, 3}){std::cout « (i * multiplicador);} // recorrerá la lista lo cual es chulísimo por que podemos tomar valores aleatorios., pero tiene que ser una lista estática definidos en tiempos de compilación.

*Int array_no_size [] {10, 12, 14, 11, 18, 15};

for (auto value : array_no_size){ //al no tener el tamaño se hace el for así.

```
    std::cout « value « std::endl;}
```

Si queremos modificar el valor del array se le pasa por referencia:

```
for (auto& value : array){value *= 2;}
```

WHILE: `while (i < 100) {instrucciones; i++;}` //la variable i debe ser declarada fuera del while.

DO WHILE: `do{instrucciones; i++;} while (i < 100);` //se hace antes de entrar en la condición

la variable i tambien se inicializa fuera del do while `(size_t i = 0;)`

---

break y continue: `break;` saldrá del loop en una determinada situación. `continue` seguirá bajo otra condición.

ARRAYS

```
int numeros [10] {1,2,3,5,6,7,8,9,10}; //ver que en c hay que incluir el =
int numeros [10] {1,2}; //se inicializa numeros[0] y 1 a 1, 2 pero el resto lo hace a cero.
int numeros [10] {}; //inicializa todos a cero.
int numeros[] {1,2,3,4,5,6,7,8,9,10,11}; //el compilador sabe el tamaño por el numero de elementos
```

pero, si queremos sacar el tamaño se hace con `std::size(numeros).`

```
    for (size_t i{0}; i < std::size(numeros); ++i){,,,};
```

previamente era con `sizeof(numeros) / sizeof(numeros[0]);`

Los arrays de caracteres se pueden imprimir como en C de printf asi:

```
char msg[5] = “hola”; //msg[4] = ‘\0’ automaticamente.
std::cout « msg; //imprime hola.
```

PERO NO PODEMOS HACER ESTO:

```
char a = “hola”;
a = “adios”; //NO SE PUEDE por que siempre ha de apuntar al inicio de “hola” que ha sido declarado.
```

Si se quierre reasignar, han de usarse punteros:

```
char *a = “hola”;
a = “adios”; //esto está bien. pero perdemos “hola” y se pierde memoria.
```

RAW C-Strings

```
char* str = R”(
gato
perro)”; //Mostrará gato y salto de linea debajo perro. La R tiene que ser mayúscula
```

MANIPULACION C-STRINGS

Utilizan #include <cstring>

std::strlen(str); //numero de caracteres del string. No cuenta \0. sizeof cuenta el \0 PERO SI NO ES UN PUNTERO. ya que si lo es, sizeof dará el tamaño del puntero (8 bytes). strlen del puntero funcionará.

std::strcmp(str1, str2); //Compara 2 strings. Hasta 1º char differente -1 ó 1. = 0 si todo igual

std::strncmp(str1, str2, n); //igual q strcmp pero compara hasta 1 "n" número. -1, 1 diferente.

std::strchr(char *str1, char ch); //devuelve puntero a 1ª aparicion de ch en str. Si no Null.

std::strrchr(str, 'a'); //lo mismo pero devuelve la última posición donde esté. Si no, nullstr.

std::strstr(str, target); //devuelve posición puntero donde está target en str. Null si no.

---

std::strcat(dest, src); //acoplará lo q esté en src, en dest. Tiene que ser dets suficiente grande

std::strncat(dest, source, n); //Lo mismo pero concatenará hasta el n carácter.

std::strcpy(dest, source); //copiará lo de source en dest. Tiene que ser suficiente grande dest.

std::strncpy(dest, src, n); //copiará n caracteres de src a dest.

PUNTEROS

Se puede inizializar punteros como:

int* puntero{nullptr}; // podria funcionar NULL en vez de nullprr pero no sería seguro. NULL = 0 que es un entero, nullptr ya que no es un valor entero. En C solo se puede NULL.

---

char* msg = "Hola Mundo!"; //ERROR de compilación. Es una cadena literal

const char* msg {"Hola Mundo!"}; // sí compila! Apunta al primer carácter 'H'. sin const no compilará por que "Hola Mundo!" es un literal (creado en memoria de solo lectura "estática") ya que si hacemos char a[] = "Hola Mundo!" eso es una copia de ese literal, pero char* msg está dirigiendose a esa memoria de solo lectura, por lo que no puede ser cambiada, y por ello se pone el const, para que compile.

El que marca qué es una cadena literal son las comillas dobles""

std::cout « *msg; // imprimirá la H solo.

---

const int* p; //Puntero a constante (entera). puede almacenar otra direccion pero no modificar la variable a quien apunta. int no tiene por que ser const. int a; p = &a // es correcto. Pero...

const int a; int *p = &a //da error por que podriamos modificar el valor de const int a.

int* const p; //Puntero constante de int. No puede apuntar otro lado pero si modificar valor de la variable.

const int* const p; //puntero constante a constante (de int) no puede apuntar a otra dirección ni puede modificar el valor.

---

char *str = “Hola Mundo”;

std::cout « “direccion memoria almacenada en str: “ « (void*)str; //tiene que ir los () tambien.

por que si se pone « str, imprimirá “Hola Mundo”. void *str sería llamar a un puntero void. y no es.

ARITMÉTICA CON PUNTEROS

Se puede hacer puntero++ y pasará numero de bytes de lo que es. (char = **ptr++** = 1byte más; int = Ptr++ = 4 bytes).

Se puede hacer diferencia entre punteros para obtener la distancia y dará el numero de bytes dependiendo de que tipo de puntero es. (char 1 byte, int 4 bytes). Es decir dará como resultado el numero de ELEMENTOS que los separa. No el numero de bytes.

Hay un tipo que como size_t almacena la diferencia: std::ptrdiff_t

std::ptrdiff_t variable = ptr1 – ptr1; //PUEDE SER NEGATIVO. Es necesario para abordar tamaño de punteros en 64bits y 32 bits que pueden provocar desbordamiento si no.

El sizeof de ptrdiff_t es 8bytes permitiendo usar mucho espacio entre arrays.

RESERVAR MEMORIA

int *array; array = new int[numero de enteros a reservar]; //reserva mem en heap de arrays

delete[] **array;** //libera la memoria de arrays CON []

array = nullptr;

LOS PUNTEROS ASIGNADOS DINAMICAMENTE LA MEMORIA NO PUEDEN TENER std::size(array), aunque los inicialices como

double *temperaturas = new double[size]{10,0,20,2,30,1}; //Para ello tendrás q pasarle el “size”

TAMPOCO SE PUEDEN HACER RANGE LOOPS:

for (auto elem : array){std::cout « elem;} //no se puede.

esto es por que “array” no es un array es un PUNTERO que almacena una array.

Para punteros: int *ptr; ptr = new int; //no se usan los corchetes con punteros.

delete **ptr;** //no se usan los corchetes en new, asi que aquí tampoco.

ptr = nullptr; //la memoria se desasigna, pero todavía puede tener valor. Aquí la borramos.

---

int *ptr = new int(32); // se le puede defererenciar desde la inicializacion para que no tenga basura.

delete ptr; ptr = nullptr;

ptr = new int (55); //esto es válido. el delete no borra la DECLARACIÓN del puntero creado.

Habría que borrar la memoria de ese ptr, pero JAMAS se ponen dos delete

```
delete ptr;
delete ptr; //MAL el segundo delete!!!
```

Puede parecer chorra.. pero si hacemos delete de otro puntero que apunta a la MISMA dirección de momoria que otro puntero que hemos hecho delete. Al borrar ese primer puntero, no haría falta borrar el segundo!!

```
int *p1 {new int {32}};
int *p2 = new int (75);
p2 = p1; //misma dirección de p1, pero la cagamos por que ya no podemos liberar memoria de p2
delete p1;
delete p2; //aquí la hemos cagado por segunda vez.
```

Podemos hacer:

```
int *p1 = new int;
int *p2 = p1; delete p2; //esto es valido y libera toda la memoria.
```

En el caso de fallar la reserva de memoria se puede hacer dos cosas:

1. TRY – CATCH (excepciones)

```
for (size_t i{},; i < 100000000000 ; ++i){
  try{int * lots_of_ints { new int[10000000]}};
  catch(std::exception& variable){ std::cout « “excepcion: “ « variable.what();}}
```

Lanza excepciones que es herramienta para ver el fallo que lo saca por el variable.what() como mensaje.

2. STD::NOTHROW

```
for (size_t i{}; i  < 100000000000; i++){
  int*  lots_of_ints { new(std::nothrow) int[10000000]}; //No lanza excepcion
  if (lots_of_ints == nullptr){std::cout « “Memoria fallida”}
  else ….. // se parece al método de C (if (!lots_of_ints).
```

Si no le ponemos el std::nothrow, en programa nofuncionaría porque lanzaría una excepción de tipo std::bad_alloc y no le asignaría el nullptr al puntero, por lo que al final el programa crashearia.

LOS ERRORES EN LINUX cuando se devuelve un return 1; se pueden ver desde la terminal como:
echo $ (enter) y devolverá el valor de retorno, del return, que es 1 si da un error (tienes que poner return 1; en caso de error

std::string

Se utiliza incluyendo #include <string>

```
std::string variable = “hello world”
std::string var1 {variable}; //var1 = hello world
std::string var2 {“hello world”, 3}; // var2 = hel
std::string var3(5, ‘a’); //var3 = aaaaa
std::string var4 {variable, 6, 5}; //var4 = world. 6 = indice inicial a partir, 5 = num de chars cpy.
```

No tiene problema en añadir la cantidad de texto necesario que sea :

```
variable = “hello world, ya ves que puedo poner lo que me d gana”; //sin problema de memoria
```

con punteros esto:

```
const char *variable = “hello world”;
variable = “hello world, ya ves que puedo….”; //lo haría pero gastaría memoria perdiendo el primer “hello world”. por que  “variable” apunta a una nueva dirección para albergar el texto más largo.
```

RAW STRINGS LITERALS (escribir lo que se quiera)

```
std::string lista {R”(
limpiar
colada
comprar comida
)”}; //lo saca como se crea con sus saltos de linea.

std::string directorio {“\”C:\\temp\””}; //“C:\temp”. Usamos caract.escape para poner \ y “.
std::string directorio {R”(“C:\temp”)”}; //No hace falta poner caracteres de escape con la R de raw.
std::string str {R”---(comillas “(parentesis)”)---”}; //comillas “(parentesis)”. para meter “( y )”. pueden ser --- o *** o japo, cualquier string, pero tiene que terminar con el mismo “japo” al final
```

---

std::string_view

#include <string_view>

Al igual que existen las variables de refererencia (char& int&, etc..) que no hacen copia de lo que se refieren, también existen los string_view, que en este caso sí se pueden usar para c-strings (las referencias no a no ser que sea const.), pero la diferencia es que los string_view no pueden ser alterados. Son constantes y su valor se almacena en la memoria estática (la de solo lectura. const int num = 32, o variables globales). Por eso no puede ser alterada.

```
void funcion (char texto[]);
char origen[] = “hola mundo”; //es un literal y por lo tanto constante (solo se puede cambiar por origen[0] = ‘C’; por que ese es una copia en el stack).
funcion (origen); //origen será copiado a texto[] en la función ocupando el doble de memoria.
```

Para que sea una sola vez la memoria y no copia:

```
void funcion (std::string_view texto){std::cout « texto;};
```

const char origen[] = “hola mundo”; //Tiene que ser const. Ocupa en memoria estatica, pero...

funcion (origen); //…en la función no será copia, sino como una referencia. Un puntero a la &origen con otro puntero con el tamaño de dicho c-string, por lo que se ahorra memoria.

std::string_view texto = “Hola Mundo”; // aquí nos ahorramos crear un char a[]{“Hola Mundo”} que contendría también el “Hola Mundo” en la memoria estática en los dos casos, PERO en la segunda, crearía en el stack la variable de tipo char a… ocupando más memoria que “texto” del string_view, que sería creada también en la memoria estática solo.

char array[]{‘H’,’o’,’l’,’a’}; //sin tener \0

std::string_view sv {array, std::size(array)}; //se necesita pasar el tamaño también ya que no \0

---

const char *texto = “Hola Mundo!”;

std::string_view sv1 = texto;

sv1.remove_prefix(5); //borra “Hola “ y queda “Mundo!”. NO ALTERA texto= “Hola Mundo!”

es solo como una VENTANA que la achicas para ver menos (O más).

sv1.**remove_suffix(1);** //dejaría de VER el carácter !

al modificar el view con suffix o prefix no se podría retornar al origen (ver todo) pero se puede crear un nuevo std::string_view sv2 = texto;

---

std::string_view texto = “hola mundo”;

texto.data(); //hola mundo. Se puede usar ,data()

texto.remove_prefix(1);

texto.data(); //ya no se puede. MAL. por que ha sido modificado.

No solo .data() sino cosas como texto.front(), texto.length(), etc.. se puede usar. Si no se acorta el view con remove_suffix o remove_prefix. Tampoco con arrays no \0

concatenar std::string

std::string str1 = “Hello”;

std::string str2 = “ World”;

std::string msg = str1 + “ my“ + str2; // Hello my World.

tambien con APPEND:

std::string msg = str1.append(str2); //Hello World

std::string msg = str1.append(5, ‘?’); //Hello?????

std::string msg = str1.append(str2, 1, 3); //Hello orl. 1 = indice; 3 = num chars to copy.

CONCATENAR NUMEROS (como itoa)

std::string msg = str1 + std::to_string(67); //Hello67. Si ponemos std::string(67) será HelloC

TRANSFORMAR EN NUMEROS UN STRING (como atoi)

```
int var = std::stoi("32"); //lo transforma en Entero 32, desde el "32" que es un string.
long var = std::stol("-32.454"); // var = -32l como numero long.
long long var = std::stoll("-32.454"); //-32ll
float var = std::stof("23.45"); //23,45f
double var = std::stod("23.45"); //23,45d
long double var = std::stold("34.432"); //32,45ld
unsigned long var = std::stoul("32.45"); //32,45l
unsigned long long var = std::stoull("32.34"); //32,34ll
```

ERRORES!!

```
std::string str3{"Hello" + "World"}; //ERROR!!!!
std::string str3{std:string{"Hello" + "World"}; //OK!
 std::string str3{"Hello" "World"}; //OK
```

También se puede evitar el ERROR con esto:

```
using namespace std::string_literals; //para usar el sufijo s que determina que es un std::string.
std::string str3{"Hello"s + "Wold"}; //al añadir esa 's' "Hello" pasa a ser std::string{"Hello"}
```

```
std::string var = "Hello";
var += ','; // Hello,
```

PERO

```
var += ',' + ' '; //segundo es un espacio → HelloL por que , es 44 ascii y ' ' es 32. Su suma es 76 por lo tanto en ascii 76 = L
(var += ',') += ' '; // Hello, (y el espacio incluido).
std::string var2 = var + ',' + ' '; //si funcionará.
```

Modificar un std::string

```
std::string str = "Hola Mundo";

str.insert(1,2, 'C'); // = HCCola Mundo. 1 = index; 2 = cuantas veces repetido
str.insert(5, "mamon"); //Hola mamonMundo. 5 = indice donde queremos meterlo.
str.inser(5, "mamon", 3); //Hola mamMundo. 3 = cuantos chars de lo que queremos insertar
str.insert(5, "eres un mamon de tomo", 7, 5); // Hola mamonMundo. 7=quitas "eres un", 5 =mamon
```

---

str.erase(6, 3); //Hola Mo. 6 = index a partir de donde vamos a borrar. 3 el numero chars a borrar.

str.push_back('!'); //Hello World! añade el carácter al final.

str.pop_back(); //borra el último carácter = Hola Mund

str.clear(); //borra el contenido dejándolo nulo pero con capacidad de (15?)

REEMPLAZAR

std::string str1 {"Encontrando Nemo"}; std::string str2 {"Buscando a"};

str1.replace(0, 12, str2); // Buscando a Nemo. 0 = index; 12 = "Encontrando "

std::string str3{"los niños estan para jugar luego"};

str1.replace(0, 12, str3, 16, 10); //Encontrando Nemo para jugar. 16=index str3; 10 num de chars

---

COPIAR:

std::string str {"quiero una copia"}; char dest[10]{}; //dest inicializado a \0

str.copy(dest, 5, 11); // dest = copia. 5 = num chars a copiar; 11 = index (inicio de "copia")

al copiarlo a un char dest[] no terminará en \0 por eso lo inicializo a \0 en la declaración dest con {}

std::strcpy(dest, a.c_str()); //necesita pasarle dos punteros. c_str() devuelve el puntero de a.

---

std::string str {"Hola"}

str.resize(8); // str = Hola\0\0\0 \0

str.resize(8,'f'); // str = Holaffff\0

str.resize(4); //str = Hola\0

CAMBIAR SWAP

str::string a {"hola"}; str::string b{"adios"};

a.swap(b); //a = adios; b = hola. internamente cambia los punteros hacia donde apuntan. No copia

ACCEDER ELEMENTOS en std::string

Para acceder dentro e iterar, tenemos que saber la longitud de la cadena std::string con var.size()

std::spring var = "Hello World";

for (size_t i{}, i < var.size(); i++) {std::cout « " " « var[i]}; //Podemos iterar elemento a elemento.

for (char valor : var){std::cout « " " « valor;} //range loops ok.

std::cout « a.at(5); //imprime 'o'. la diferencia con a[5] es que asegura .at que está en el rango y si no lanza una excepción std::out_of_range.

a.at(0) = 'C'; // = a[0] = 'C'; → Cello World.

char& char_inicio = a.front(); //H . Defino char_inicio como referencia para cambiarlo

char char_final = a.back(); //d

char_inicio = 'C'; //si cambiará y será Cello World, por que char_inicio es UNA REFERENCIA

char_final = 'F'; //no lo cambiará por que char_final es UNA COPIA.

TRANSFORMANDO A PUNTEROS (.c_str(), .data())

std::string str1{"Hello World"};

char *aux = str1.data(); //ahora aux apunta a la H. Podemos hacer un cout de aux.

aux[0] = 'C'; //aux = Cello World.

const char *aux2 = str1.c_str(); //devuelve un puntero a constante por eso ha de ser const char*. No se puede modificar por lo tanto. Con .data() si

---

BUSQUEDA TEXTO EN STD::STRING

std::string str{"hola mundo"};

str.find("nd"); //devuelve indice donde lo encuentra: 7. Si no devuelve std::string::npos

if (str.find("nd" == std::string::npos) {std::cout « "no encontrado"}; el numero npos es el mismo.

size_t num = -1; //num = 18446744073709551615 = std::string::npos . Siempre es el mismo valor.

str.find("o", 4); //segundo o de "mundo" Empieza a buscar a partir index(4). Sin 4 es el 'o' en hola.

Tamaño std::string

std::string a = "Hola Mundo";

a.length(); //igual a a.size()

a.empty(); // dice si está vacio o no con un 1 o un 0 (std::boolalpha lo dará con true o false)

a.max_size(); //da el tamaño máximo de caracteres que puede tener un std::string.

a.capacity(); //da el tamaño max. que puede poner sin crear otro dinamico en memoria.

std::string b; b.capacity(); //da 15, ya que crea por defecto 15 espacios para rellenar

a.reserve(100); //reserva 100 espacio que se necesita por que está limitado por ::capacity().

a.shrink_to_fit(); //cambiará la capacidad a el tamaño actual del array contenido en std::string

en el caso de haber reservado 100 antes, y hacer el shrink no cambairá a 10 de "Hola Mundo" sino a 15, que es el minimo que genera en un nuevo std::string b;

Comparar std::string

Se hace mediante operadores de comparación > < != y ==

```
std::string a = “Patata”; std::string b = “Batata”. if (a > b); //es true por que es mayor la P que la B.
```

```
std::string a = “Patata”; std::string b = “Batata”
a.compare(b); // +1 por que P es más grande que B
std::string a = “Hello”; std::string b = “Hello World”
a.compare(6,5,b); // comparará el World sacado de b (6 = index , 5=world tiene 5 chars)
a.compare(“Hello”); // daría 0 por ser igual
```

CLASES

Ejemplo de Clase dentro de MyClass.hpp:

```
class MyClass {
    public:
        MyClass(parametros); //constructor
        ~MyClass(void); //Destructor
    void  funcion(parametros);
    typo  getFuncion(void) const; //const es para no modificar el get
    void  setFuncion(parametro);

    private:
        type  _var1;
};
```

Las clases por defecto si no se pone public o private, por defecto será PRIVATE. Una Estructura es como una clase, con la única diferencia de que si no se especifica todo es PUBLIC.

Como hay parametros PRIVATE solo se podrian acceder a los mismos desde la misma clase, pero para acceder desde fuera para leerlos están los GETTERS (getFuncion) y los SETTERS (setFuncion) para modificarlos de lal manera que se controla dentro de las funciones como se modifican dichos valores privados.

la implementación de las funciones se hace dentro de otro archivo cpp MyClass.cpp:

```
MyClass::MyClass(parametros){
    _var1 = parametros; //si coincide con el tipo claro...
```

```
}
MyClass::~MyClass(void){
    std::cout << "Llamado el destructor" << std::endl;
}
MyClass::funcion(void){....}
typo MyClass::getFuncion(void) const{
    if (cliente) {return _var1;}  //se puede controlar quien puede leer el valor privado...
void MyClass::setFuncion(std::string _var1){
    if (cliente){
        this->_var1 = _var1; //el this refleja que es el parametro privado. A la  derecha del igual
    es el parametro que recibe. En caso de que sean los dos iguales.
    }
}
```

Cuando se declara en un main una variable de esa clase (MyClass variable(parametros)) se llama automaticamente al constructor sin hacer nada, y cuando se hace un delete o un return 0, por finalizacion de la función se llama automaticamente al destructor. Siempre el destructor último a ser llamado será el primer objeto creado, ya que puede haber dependencias de ese primer objeto y de esa manera se garantiza no tener fallos.

NEW - DELETE EN CLASES

```
class Zombie{
    public:
        Zombie(std::string name){
            _name = name;
        }

    private:
        std::string _name;
};

int main(){
    Zombie *z;
    Zombie *horde;
```

```
    z = new Zombie("Bartolo");
    horde = new horde[100]; //ESTE NO FUNCIONARIA pero es un ejemplo para ver como se declararia. No funcionaria por que debe recibir el parametro del nombre en cada constructor y al ser un array no puede hacerlo, para ello habria que hacer un SETTER y declararlo con un for uno a uno de los 100.
    ...
    delete z;
    delete[] horde;
    return 0;
}
```

PUNTEROS A FUNCIONES

Imaginemos la clase:

```
class Harl{
    public :
        Harl(void);
        ~Harl(void);

    private :
        void debug(void);
        void info(void);
        void warning(void);
        void error(void);
};
```

Para declarar un array de punteros a las funciones:

```
    void (Harl::*funciones[])() = {&Harl::debug, &Harl::info, &Harl::warning, &Harl::error};
//ver que no es &Harl::debug()... es decir se omite los () por que no son funciones sino punteros.
```

Para llamar a las funciones seria:

```
    (this->*funciones[index])();
```

SOBRECARGA DE OPERADORES

Es como la sobrecarga de operadores (funciones) en donde el mismo nombre de la función sirve para diferentes parámetros, pero en este caso para operadores.

Si nosotros tenemos una clase Point:

```
Pointpoint1(3, 4);
Pointpoint2(0, 2);
Pointpoint3(point1 + point2) //esto dará error si no esta definido la sobrecarga de operadores
```

Si no está definida ese operador suma daría error, por lo tanto hay que definirlo, por que no sabría como sumar dos objetos (no son numeros). Por lo tanto lo definimos:

```
Pointoperator+(const Point &right_operand){
    return Point(this->x + right_operand.x, this->y + right_operand.y);
} //Si está declarado dentro de la clase como miembro

Pointoperator+(const Point &left, const Point &right){
    return Point(left.x + right.x, left.y + right.y);
} //Si está declarado FUERA de la clase como NO miembro
```

El que está dentro de la clase como miembro, aunque solo recibe 1 parametro, está implicito ese primer parametro que es el de this-> y el segundo es el del parámetro que se incluye. Mientras que el operador sobrecargado si está fuera de la clase, entonces hay que definir los dos parametros.

en si un point1 + point2 es como poner esto: point3.operator+(point1, point2)

*IMPORTANTE: SIEMPRE ES (operator+) es decir tiene que ser la palabra tal cual, que puede ser tambien operator-

SOBRECARGA DE OPERADORES - Subscript operator

El operador subscript es (operator[]) sirve para acceder por valores de índice a los miembros de una clase. Por ejemplo para la clase Point:

```
Point punto(1, 4);
```

dentro de la clase:

```
int operator[](size_t index) const{
    if (index == 0)
        return (this->x)
    if (index == 1)
        return (this->y)
}
```

Si quisieramos sustituir un valor de tal manera que:

punto[0] = 2;

punto[1] = 7;

Entonces tenemos que definir el operador como:

```
int &operator[](size_t index) { //ponemos el & y quitamos el const
    assert( (index == 0) || (index == 1) ); //hay que #include <cassert>
    return(index == 0) ? this->x : this->y
    }
```

Es como utilizar un getter para tener un Setter. Con el & se modifica el valor dentro.

SOBRECARGA DE OPERADORES - Output Stream Operation

Podemos hacer con una class Point

```
Point punto1(3,4);
std::cout << punto1;
```

Solo se puede hacer si se usa la sobrecarga del operador <<. Pero para que funcione con el std::cout << tiene que ser fuera de la clase:

```
class Point{
    public:
        friend std::ostream &operator<<(std::ostream &out_msg, const Point &point);
//friend hace que pueda ser accesible desde fuera como un getter la función definida según la declaración, pero friend es C++11.
        Point(void);
        ~Point(void);
```

```
    private:
        double x{}; //lo inicializa a 0.0
        double y{};
};
inline std::ostream &operator<<(const std::ostream &out_msg, const Point &point){ //inline hará de protección para no tener múltiple definiciones de este operador cuando se llame en cada .cpp el include de este .hpp. No proteje para eso el #ifndef HEADER_HPP .....
    out_msg << "( " << point.x << ", " << point.y << " )" << std::endl; //si no estuviera el friend, no se podria acceder a los parámetros privados.
    return (out_msg);
}
```

Se podría meter la sobrecarga dentro de la clase asi en parte pública y por lo tanto no necesitar el friend:

```
std::ostream &operator<<(const std::ostream &out_msg);
```

Pero al estar implícito el primer parámetro "this" miembro de la clase, el sistema de salida tendría que ser:

```
punto1 << std::cout; //tendríamos que invertirlo por lo que no se suele ver declarado dentro d la función
```

SOBRECARGA DE OPERADORES - Input Stream Operation

Se puede escrbir de std::cin >> a un objeto con la sobrecarga de >>

```
inlinestd::istream &operator>>(std::ifstream &insert_point, Point &point){
    double x;
    double y;
    std::cout << "introduzca coordenadas del punto" << std::endl;
    std::cout << "orden x, y separado por espacios : ";
    insert_point >> x >> y;
    point.x = x;
    point.y = y;
    return insert_point;
}
```

Y ahora se podría hacer

```
std::cin >> punto1;
```

HERENCIAS DE CLASES

Lo que hace es heredar propiedades de una clase padre.

```
class Persona{
    public:
        Persona();
        Persona(std::string first_name_param, std::string last_name_param);
        ~Persona();
    Private:
        std::string first_name{"Misterioso"};
        std::string last_name{"Persona"};
};
```

La clase que hereda será:

```
class Jugador : public Persona{
    public:
        Jugador() = default;
        Jugador(std::string game_param);
        ~Jugador();
    private:
        std::string m_game{"none"};
};
std::ostream &operator<<(std::ostream &out, const Player &player){
    out << "game: " << player.m_game << "name: " << player.get_first_name()....
}
```

Y esta clase que hereda no puede acceder a la parte privada de Persona. Para ello deberíamos crear en la parte pública de Persona dos getters (uno para first_name y otro para last_name

HERENCIAS DE CLASES / Protected

Es un nuevo tipo de estiqueta en la clase:

```
class Persona{
    public:...
    protected:
    private:
};
```

Todo lo que está en protected es algo como privado, PERO puede ser accedido desde las clases que heredan. Sirve para proteger, como private pero que funcione en las herencias

HERENCIAS DE CLASES / Herencias Protected o Private

Cuando se hace una herencia, no solo puede ser `: public ClasePadre`

Puede ser de dos tipos:

1. `class Heredada : protected ClasePadre`

    En la clase heredada, todo lo que era publico del padre se convierte en Protected

    Todo lo que era Protected seguirá siendo Protected.

    Todo lo que era Privado seguirá siendo Privado

2. `class Heredada : private ClasePadre`

    En la clase heredada, todo lo que era publico del padre se convierte en Private

    Todo lo que era Protected se convertirá en Private

    Todo lo que era Privado seguirá siendo Privado

El problema es que una vez se sube un escalón para proteger en herencia no se puede volver atras, y si tenemos una herencia que es private, y por lo tanto todo es private, al hacer una clase heredada de dicha clase, aunque le pongamos `: public Heredada` todo seguirá siendo private.

PERO!! podemos RESUCITAR un método usando USING

```
class Felino : private Animal{
    public:
        using Animal::respirar; //siendo void respirar(); en público de Animal
        void maullar();
};
```

Con eso ya se tranforma en público de nuevo y es válido en C++98. Podriamos hacer using en protected: y volveria a ser protected.

si la funcion a la que llama (en el ejemplo "respirar") está sobrecargada, resucitará a todas las sobrecargas de funciones.
NO PODRIAMOS HACER CON USING acceso a una parte PRIVATE de Animal. Por que es la exclusión que tiene.

PERO SÍ que podriamos hacer public algo que fuera Protected. Es decir la regla, es "todo lo que fuera accesible, se puede reconvertir a nuevo acceso.

HERENCIAS DE CLASES / CONSTRUCTORES NO DEFAULT

Si tenemos constructores no default que reciben parámetros:

```
class Animal {
private:
    std::string nombre;
    std::string raza;
public:
    Animal(const std::string& n, const std::string& r) : nombre(n), raza(r) {}
};


class Gato : public Animal {
private:
    int edad;
public:
    Gato(const std::string& n, const std::string& r, int e)
        : Animal(n, r), edad(e) {}
};
```

La llamada al constructor dentro de la clase hijo, se ha de hacer por lista, ya que si se llama Animal(n, r) dentro de las llaves, es como hacer una copia que no queremos.

HERENCIAS DE CLASES / CONSTRUCTOR DE COPIA

```
class Animal {
public:
    Animal(const Animal& other) {
        // copiar datos
    }
};
```

```
class Gato : public Animal {
public:
    Gato(const Gato& other)
        : Animal(other) // llama al constructor de copia de Animal
    {
        // copiar datos específicos de Gato
    }
};
```

HERENCIAS DE CLASES / constructores heredados

Podemos hacer que el constructor del heredado no sea el suyo sino el del padre con Using (esto es de C++11):

```
class Animal {
private:
    std::string nombre;
    std::string raza;
public:
    Animal(const std::string& n, const std::string& r) : nombre(n), raza(r) {}
};

class Gato : public Animal {
public :
    using Animal::Animal; //herencia de los constructores
private:
    int edad;
};
```

Los demás parámetros de la clase hija que sea suyos, se inicializarán a cero de echo, si el constructor de Gato antes era:

```
Gato(const std::string& n, const std::string& r, int e) : Animal(n, r), edad(e) {}
```

Y al ser llamado en un main como Gato gato1(“Jhonsy”, “comun”, 10); tenía que admitir 3 parámetros, o tendríamos un error de compilación. En el constructor heredado con Using, podremos pasarle dos parámetros solo Gato gato1(“Jhonsy”, “comun”), y la edad se pondrá como valor basura si no tiene una definición por defecto.
Al heredar los constructores con Using, pillará todos los que haya, ya sean por defecto o custom. PERO NO heredará los constructores de copia.

Si tenemos una herencia de constructores en este caso ahora, Gato gato1("Jhonsy", "comun", 10); dará un error de compilación ya que no existe la definición con 3 parámetros

HERENCIAS DE CLASES / dynamic cast / acceso a métodos derivados

Si queremos acceder a un método derivado de una clase padre:

```
class Padre{};
class Hija : public Padre{
    public:
        void funcion(){}};
int main(void){

    ptr_padre→funcion(); //ERROR compilación
```

Esto da error de compilación por que la clase puntero es la clase base (Padre), y no la clase derivada(Hija), por lo que al compilar (estaticamente) el compilador mira si tiene ese método la clase Padre y no lo encuentra.

Para resolverlo podemos usar dynamic_cast

```
int main(void){
    Hija hija1;
    Padre *ptr_padre = &hija1;
    Hija *ptr_hija = dynamic_cast<Hija*>(ptr_padre);//convierte a objeto Hija lo q apunta el padre
    if (ptr_hija) //puede que de nullptr
        ptr_hija→funcion();
```

Esto comprueba en tiempo de ejecución si el objeto apuntado por el puntero base pertenece al tipo derivado y si lo hace (de ahí el if) ya podemos acceder a el.

También se puede hacer por referencia:

```
int main(void){

    Hija hija1;
    Padre &ref_padre = hija1;
    Hija &ref_hija    = dynamic_cast<Hija&>(ref_padre);
    ref_hija.funcion();
```

Pero al hacer esto como no devuelve un puntero no se puede comprobar. La forma de hacerlo es:

```
int main(void){
    Hija hija1;
    Padre &ref_padre = hija1;
    Hija *ptr_hija = dynamic_cast<Hija*>(&ref_padre); //convertimos la referencia a puntero
    if (ptr_hija)
        ptr_hija→funcion(); //mejor con el puntero. Este ejemplo es para ver como se convierte.
```

Hay que tener mucho cuidado ya que si a las dos clases Padre e Hija añadimos una Nieta:

```
class Nieta : public Hija{};
```

Si le hago:

```
int main(void){
    Padre *ptr_padre = new Hija();
    Nieta *ptr_hija = dynamic_cast<Nieta*>(ptr_padre);
    ptr_hija→funcion(); //esto crasheará en tiempo de ejecución
    delete ptr_padre;//leaks ya que no es destructor polimorfico (ver en polimorfismo)
```

Esto compilará sin problemas pero al ejecutar se crasheará por que el ptr_hija es un nullptr, ya que no puede acceder a esa información. Por eso hay que comprobarlo con un if para que no sea nullptr.

Pero para hacer esto sería mejor hacerlo con polimorfismo (ver más abajo…)

POLIMORFISMO

Cuando heredamos de clases, puede que queramos hacer métodos específicos para cada uno de los miembros que pertenecen a una clase padre. Así si tenemos un polígono como padre, e hijos, Círculo, Ovalo y Cuadrado queremos un método que dibuje cada uno de estos y sea llamado conforme reconozca que tipo de polígono es.

IMPORTANTE: No se puede hacer llamada a dichas funciones desde constructores o destructores, ya que en el proceso de creación de los objetos hijos, primero se hacen los superiores (para crear el nieto se hace primero el padre luego el hijo y por último el nieto) por lo que al querer llamar a un método (función) específico de un objeto derivado, no podría acceder a el, si se mete en un constructor de dicho objeto

Se puede hacer esto:

```
class Polygon {
private:
    std::string name{“”};
public:
    Polygon(const std::string& n) : name(n){};
    void draw() const{std::cout « “Dibujo un “ « name};
    std::string get_name() {return (name)};
};
class Oval : public Polygon{
private:
    float  radio_x{0.0};
```

```
    float  radio_y{0.0};
public:
    Oval(const std::string& n, const int r_x, const int r_y) : Polygon(n),
        radio_x(r_x), radio_y(r_y){};
    void draw() const{std::cout « “Dibujo un “ « get_name() « “con radios « radio_x
        « “, “ « radio_y};
};
class Circle : public Oval{
private:
    float radio{0.0};
public:
    Circle(const std::string& n, const int r) : Oval(n, r, r), radio(r){};
    void draw() const{std::cout « “Dibujo un “ « get_name() « “con radio « radio};
};
int main(void){
    Polygon    pol1(“cosa”);
    Oval       oval1(“ovalo, 2.3, 1.4);
    Circle     circle1(“circulo”, 5.3);
    pol1.draw();
    oval1.draw();
    circle1.draw();
    return (0);
}
```

Y dibujaría bien cada uno de los objetos, PERO si queremos tratarlos por punteros o referencias:

Vinculación Estática

```
int main(void){
    Polygon    pol1(“cosa”);
    Oval       oval1(“ovalo, 2.3, 1.4);
    Circle     circle1(“circulo”, 5.3);
    Polygon   *todos_poligonos[]{ &pol1, &oval1, &circle1}; //se pueden meter todos en array
    for (Polygon *i_ptr : todos_poligonos){ //c++11 CUIDADO!!
        i_ptr→draw();}
```

```
    return (0);
}
```

Este bucle for no va a dibujar el método específico de cada objeto derivado, sino los tres van a ser el del padre Polygon, por que se está llamado a un puntero tipo Polygon y no tipo Oval ni Circle. Esto es por vinculación estática que tiene el compilador.

POLIMORFISMO. Vinculación Dinámica

Para que cuando se llame a un objecto mediante su puntero o referencia padre, sepa ver cual es el método que tiene que aplicar, solo tenemos que marcar dichos métodos con la palabra virtual asi cuando ponemos en los métodos de la clase anterior:

```
virtual void draw() const{std::cout « “Dibujo un “ « name};
void draw() const{std::cout « “Dibujo un “ « get_name() « “con radios « radio_x
    « “, “ «  radio_y};
void draw() const{std::cout « “Dibujo un “ « get_name() « “con radio « radio};
```

NO HACE FALTA PONER VIRTUAL A LAS CLASES DERIVADAS, aunque tengan estas otras subclases derivadas. El padre marcado como virtual marca todo hacia abajo. Se puede poner a todas, pero no hace falta. Y SOLO HACE FALTA MARCARLO EN EL .hpp

al hacer:

```
int main(void){
    Polygon   pol1(“cosa”);
    Oval      oval1(“ovalo, 2.3, 1.4);
    Circle    circle1(“circulo”, 5.3);
    Polygon   *todos_poligonos[]{ &pol1, &oval1, &circle1}; //se pueden meter todos en array
    for (Polygon *i_ptr : todos_poligonos){ //c++11 CUIDADO!!
        i_ptr→draw();}
    return (0);
}
```

ahí sí que llamará a cada método correspondiente, aunque sean todos punteros a Polygon.

POLIMORFISMO. Object Slicing

Cuando copiamos a un objecto padre un objecto hijo, se pierden datos, ya que el padre no tiene el espacio para almacenar los datos extra. Un Polygono tiene el nombre solo, y un Ovalo, tiene ese nombre MAS los dos radios. Si hacemos:

```
Polygon   pol1 = oval1;
```

Solo la parte de oval1 del objeto Polygon se copiará a pol1. Así por eso en referencias polimórficas:

```
int main(void){
    Polygon    pol1("cosa");
    Oval       oval1("ovalo, 2.3, 1.4);
    Circle     circle1("circulo", 5.3);
    Polygon &ref = pol1;
    ref.draw();
    ref = oval1; //slicing
    ref.draw();
    ref = circle1; //slicing
    ref.draw();
```

No hará polimorfismo auque esté marcada el padre como virtual de draw(). Para ello hay que hacer:

```
int main(void){
    Polygon    pol1("cosa");
    Oval       oval1("ovalo, 2.3, 1.4);
    Circle     circle1("circulo", 5.3);
    Polygon &ref = pol1;
    ref.draw();
    Polygon &ref2 = oval1; //no slicing
    ref2.draw();
    Polygon &ref3 = circle1; //no slicing
    ref3.draw();
```

HAY QUE TENER CUIDADO por que cuando hacemos colecciones de tal manera que tenemos:

```
int main(void){
    Polygon    pol1("cosa");
```

```
    Oval        oval1("ovalo, 2.3, 1.4);
    Circle      circle1("circulo", 5.3);
    Polygon   varios[] {pol1, oval1, circle1} //slicing
}
```

Ocurre Slicing (pérdida de datos) por que los objectos albergados en la colección son COPIAS. Para evitarlo hay que hacer:

```
Polygon   *varios[] {&pol1, &oval1, &circle1} //no slicing
```

Ahí si que podria haber polimorfismo sin slicing. PERO no hay que hacerlo con Referencias:

```
Polygon   &varios[] {pol1, oval1, circle1} //esto no compilaría
```

POLIMORFISMO. OVERRIDE (cuidado!! C++11)

Si declaramos una función con otro nombre, o cometemos un error de typo, encontrar dicho error puede ser un problema a veces:

```
class Padre {
public:
    virtual void dibujar(){};};
class Hija : public Padre{
public:
    void dibujar() const {};} // diferente por que lleva const pero podria ser Dibujar con mayus
```

En este ejemplo de arriba la función tiene la palabra const en la clase hija, por lo tanto no se declaran igual y al compilar no habrá ningún problema. PERO serán dos métodos diferentes.

Para garantizar que funcione y nos muestre el error en compilación podemos añadir la palabra override (solo a partir de C++11).

```
class Padre {
public:
    virtual void dibujar(){};};
class Hija : public Padre{
public:
    void dibujar() const override {};}
```

Y con ello, al compilar le estamos diciendo al compilador que queremos que dicha función sea “machacada” por esta otra (polimorfismo) y si el compilador detecta que hay una incongruencia como dibujar() vs Dibujar(), dará error y dirá que no se puede override.

POLIMORFISMO. con Overload. Name Hiding

Si tenemos una clase con overload de métodos:

```
class Padre {
public:
    void dibujar(){};
    void dibujar(int color)};
class Hija : public Padre{};
```

Podemos hacer el Overload sin problemas aquí:

```
int main(void){
    Padre   padre1;
    Hija      hija1;
    hija1.dibujar();
    hija1.dibujar(32);}
```

Pero si hacemos polimorfismo:

```
class Padre {
public:
    void dibujar(){};
    void dibujar(int color)};
class Hija : public Padre{
    //si añadimos esta linea: using Padre::dibujar; ya no se produciría el name hiding. C++11!!!!
    dibujar(){} override;}; //este oculta (name hide) todos los métodos declarados “dibujar”
```

Por lo tanto en el main que hicimos antes, la linea última:

```
hija1.dibujar(32);}
```

dará error de compilación, ya que todo se ve anulado por el polimorfismo de la hija.

Para que funcionara tendríamos que hacer esto:

```
class Padre {
public:
    void dibujar(){};
    void dibujar(int color)};
class Hija : public Padre{
    dibujar(){} override;
    dibujar(int color) override};
```

También hay que tener en cuenta que si hacemos en la hija:

```
class Hija : public Padre{
    dibujar(){} override;
    dibujar(int color) override
    dibujar(int color, const std::string nombre}; //Nueva sobrecarga solo en la hija.
```

Este último método declarado será una nueva sobrecarga dentro de la hija.

POLIMORFISMO. Variables Estaticas

Clases pueden tener variables estáticas que son heredadas por las hijas:

```
class Padre {
public:
    Padre(){++contador};
    static int contador;
class Hija : public Padre{};
```

Si hacemos en un main:

```
int main(void){
    Padre   padre1;
    std::cout << Padre::contador; //contador = 1
    Padre   padre2;
    std::cout << Padre::contador; //contador = 2
    Hija      hija1;
    std::cout << Padre::contador; //contador = 3
```

Pero si ponemos en el main:

```
    std::cout << Hija::contador; //contador = 3
```

Aunque hereda, no es independiente esa variable y por lo tanto se incrementa también. Si queremos hacerla independiente, tendríamos que definirla como:

```
class Hija : public Padre{
    static int contador;}; //con el mismo nombre
```

y ahora sí:

```
std::cout << Hija::contador; //contador = 1
```

POLIMORFISMO. final. A partir de C++11!!!

Podemos introducir una “etiqueta” en el código para que o los métodos o las clases no puedan ser heredadas:

1. Impositibilitar herencias de Métodos:

```
class Padre {};
class Hija : public Padre{
    virtual void funcion() const final{};}; //chorrada poner virtual ya que no lo permitirá final
class Nieta : public Hija{
    void funcion() const override{}}; //dará compilador error por que funcion es final en Hija.
```

2. Impositibilitar herencias de Clases:

```
class Padre {};
class Hija final : public Padre{};
class Nieta : public Hija{}; //dará compilador error por que Hija es final.
```

POLIMORFISMO. Destructores. EVITAR LEAKS

Si tenemos unas clases:

```
class Padre {
public:
    ~Padre();};
class Hija : public Padre{
public:
    ~Hija();}
```

Si en el main hacemos esto:

```
int main(void){
    Padre *ptr_padre = new Hija;
    delete prt_padre; //fuga memoria
```

Porque al no ser polimórficas los destructores, se llamará al destructor de Padre y por lo tanto no al del Hija, y se perderá la asignación de bytes de memoria del Hija

Para solucionarlo hay que hacerlas polimóficas con virtual

```
class Padre {
public:
    virtual ~Padre();};
class Hija : public Padre{
public:
    ~Hija();}
```

y así al hacer el delete ptr_padre, se llamará primero al destructor del Hija y luego el de Padre.

POLIMORFISMO. typeid(). #include <typeinfo>

Con el polimorfismo, para debuguear si estamos en una dynamic binding (polimorfismo) o static binding (no polimorfismo) podemos usar la instrucción typeid(tipo).name()

La forma en la que funciona el typeid es la siguiente fuera de las clases:

```
if (typeid(32) == typeid(int)) do_something(); //aquí lo haría ya que 32 es un int
```

Para sacar qué tipo es un tipo en si, le metemos el método .name()

```
std::cout << typeid(int).name(); //esto dependiendo del compilador imprimirá “i” o “int”
```

Ahora, ¿Para qué nos sirve esto en objetos?

```
class Padre {
public:
    virtual funcion();};
class Hija : public Padre{
public:
    funcion();
int main(void){
    Padre *p_padre = new Hija;
    std::cout << typeid(p_padre).name() << “\n”; //dynamicBase
std::cout << typeid(*p_padre).name() << “\n”;} //dynamicDerived
```

Podemos debuguear de qué tipo es cada uno de los objetos. Si no se dereferencia el puntero, será estático (dynamicBase, por que se refiere a la base) y si defererenciamos el puntero será dynamic derivado es decir dinámico (polymórfico)

Si en el ejemplo anterior fueran sin virtual, es decir no polimórficas, en las dos obtendriamos: staticBase.

POLIMORFISMO. Clases Puras y abstractas

Puede que cuando queramos hacer una clase, la clase Padre, no queramos implementar sus métodos, ya que queremos que sean los hijos los que lo implementen. Esto se consigue así:

```
class Padre {
public:
    Padre() = default; //C++11
    virtual ~Padre();
    virtual funcion1() const = 0;
    virtual funcion2() const = 0;
```

Es decir, igualándolas a cero, pero siendo virtuales para ser dynamic binding. Esto va a ser una clase ABSTRACTA. Y solo hace falta que tenga un método virtual PURO (es decir, el método igualado a cero)

1. Método virtual puro = Tiene que ser virtual e igualado a cero
2. Clase abstracta = Tiene que tener al menos un método virtual puro.

No existen clases abstractas ni métodos puros si no son virtuales.

IMPORTANTE: Cuando se tiene una clase abstracta, no se puede crear un objeto base de dicha clase:

```
Padre *padre = new Padre(); //error de compilación
```

Pero sí se puede:

```
Padre *hija = new Hija();
```

POLIMORFISMO. Interfaces

¿Para qué sirve pues una clase abstracta? Pues aunque ocupa memoria, por que si bien no se pueden crear objetos base de una clase abstracta, sí que van implícitos en los objetos creados derivados, sirve para definir una INTERFACE en la cual todos las clases derivadas tienen que override los métodos virtuales puros definidos en la clase Padre.

IMPORTANTE: Si en las clases derivadas, no hacemos override de todos los métodos puros de la clase Padre, dicha clase derivada pasará a ser una clase abstracta también.

Dicha INTERFACE sería PURA si TODOS sus métodos son puros. Pero podría ser que alguno no lo fuera. Entonces sí podria ser una interface pero no pura (ver más abajo con inline del operator<<). Por ello no hace falta ni definir constructores ni destructores.

Pero una interface puede ayudarnos a ahorrar tiempo, si por ejemplo definimos el método de output de std::out así:

```
class Interface {
    friend std::ostream &operator<< (std::ostream &out, const Interface &operand){
        operand.stream_insert(out);
        return (out); //pero friend es c++11
```

```
public:
    virtual void stream_insert(std::ostream &out)const = 0;};
class Point : public Interface{}; //tiene que sobreescribir el metodo stream_insert
class Animal : public Interface{}; //tiene que sobreescribir el metodo stream_insert
```

Con esto ya estamos permitiendo que cualquier clase derivada aunque no tenga nada que ver, pueda utilizar el operador << para sacar datos por pantalla.

Así como ejemplo si definimos

```
class Point : public Interface{
public:
    void stream_insert(std::ostream &out)const override{
        out << “Point: x=” << m_x << “ y=” << m_y;}
private:
    double  m_x;
    double  m_y;};
```

De esta forma nos ahorraríamos la declaración de la sobrecarga del operador << en cada clase.

PARA hacerlo sin friend (c++11) y compatible con c++98 (maldito 42) haríamos esto:

```
class Interface {
public:
    virtual void stream_insert(std::ostream &out) const = 0;
    virtual ~Inteface();
    //método publico para que el operador global pueda usarlo
    void print(std::ostream &out) const { stream_insert(out); }
};
//Operador global inline llama al método público
inline std::ostream &operator<<(std::stream &out, const Interface &obj){
    obj.print(out);
    return (out);
}
class Point : public Interface{
public:
    Point(double x, double y) : m_x(x), m_y(y){}
    void stream_insert(std::ostream &out) const override{
        out << "Point: x=" << m_x << " y=" << m_y;}
private:
```

```
    double m_x;
    double m_y;
}
```

Y se podria usar con cualquier otro tipo de clases aunque no tengan nada que ver:

```
class Animal : public Interface {
public:
    void stream_insert(std::ostream &out) const override {
        out << "Soy un animal";
    }
};
//////////////////
class Avion : public Interface {
public:
    void stream_insert(std::ostream &out) const override {
        out << "Soy un avion volando";
    }
};
///////////////
int main() {
    Point p(1,2);
    Animal a;
    Avion v;

    std::cout << p << "\n" << a << "\n" << v << std::endl;
}
```

EXCEPCIONES

Sirven para debuguear o más bien controlar posibles errores en el código y no cometer Segfault

EXCEPCIONES. Try-Catch

Es el método general y sirve desde C++98. La teoría es que se generan dos bloques. Uno try en donde se prueba el código problemático y en el habrá un if, que lanzará un throw (una excepción) y lo pillará el bloque del catch, que mostrará el error

```
try {   //codigo conflictivo
    if (//condición a comprobar)
        throw 0; //0 o lo que sea
}catch(variable lanzada){//mostrar error}
```

ejemplo:

```
int num{0};
try{if (num == 0) //se meterá seguro aqui
    throw 0; //lanza la excepción
num++; //nunca llega aquí
}catch(int ex){ //ex es la excepción lanzada
    std::cout << “Algo fue mal con codigo ” << ex << std::endl;
```

MUY IMPORTANTE!! todo aquello que se declare en el try si es local se pierde y si se reserva memoria también y habrá leaks.

1. Variables locales destruidas fuera del try al alcanzar el catch

```
try{
int num = 0;
int *p_num = &num; //variable puntero local del try
while (1){
    if (num == 5)
        throw p_num;
    (*p_num)++;
}
}catch (int *ex){ //se recoge un puntero colgante ya que se destruye al salir del try
    std::cout << “el error es: ” << *ex << std::endl;} //indefinido por puntero colgante
```

Sin embargo cuando se hace:

```
try{
int num = 0;
while (1){
    if (num == 5)
        throw num;
    num)++;
}
}catch (int ex){ //aqui aunque se borre del stack el num, se ha realizado una copia. Perfecto.
    std::cout << “el error es: ” << ex << std::endl;} //sin problemas por que es una copia el throw.
```

2. Reservas de memoria en el try al alcanzar el catch

```
int num{0}
try{
    int *p_num = new int; //lo que se reserve dentro hay que liberarlo dentro del try.
    *p_num = num;
    if (*p_num == 0){
        delete p_num;//si no se incluye esta linea habría leaks.
        throw 0;
    }
}catch (int ex){
    std::cout << “el error es: ” << ex << std::endl;}
```

Hay que liberar dentro del if, por que si no habría leaks, ya que no hay manera de hacer una liberación de memoria fuera del try al ser el puntero local.

3. Throw en funciones anidadas.

Podemos lanzar catch dentro de funciones anidadas, pero hay que tener cuidado en no perder memoria:

```
f1(){f2(); …;} //no se ejecutará los ...
f2(){f3(); …;} //no se ejecutará los ...
f3(){throw 0; …;} //no se ejecutará los ...
int main(){
    try{
        f1();
    }catch(int ex){}
```

```
}
```

En el código de arriba, en el main llamamos a la funcion f1(), esta a su vez llama a f2 y f2 llama a f3, pero en f3, lanzamos una excepción, por lo que TODO lo que esté detrás de dicha excepción no se ejecutará (los …). Tratará de buscar un catch en f2 y no lo encontrará, por lo que sus … después de la llamada a f3 en f2 no se ejecutará y pasará a buscar el catch en f1, y como no lo encontrará, no ejecutará el resto de … de f1 a partir de la llamada a f2(). Se irá al main y verá el catch y terminará, PERO si habíasmos hecho liberación de memoria en alguno de los … no se ejecutarán y por lo tanto habrá leaks.

SE PUEDE hacer catch en cualquier parte de la anidación de funciones, PERO es mejor hacerlo cerca del throw en vez de lo más fuera, por que más probabilidades de encontrarnos errores:

```
f1(){f2(); …;} //SI se ejecutará los ...
f2(){f3(); …;} //SI se ejecutará los ...
f3(){
    try{throw 0;
        …; //NO se ejecutarán los ...
    }catch(int ex){};
    …;} //SI se ejecutará los ...
int main(){f1();}
```

IMPORTANTE: Si hay un throw (sin su try incluido por que si no, no compila), siempre tiene que haber un catch, o si no habrá un crasheo del programa.

Si se lanza una excepción con un objecto:

```
try{
    MiClase   a;
    throw a; //tambien puedo borrar la linea de arriba y hacer throw MiClase();
}catch(MiClase ex){...}
```

Dicha clase, tiene que tener un constructor de copia, ya que se hará una copia al lanzar la excepción. Si el constructor de copia es borrado, protegido o privado no se lanzará la excepción, por que como hemos dicho, todo lo que esté dentro del try perecerá al ser lanzada la excepción y si no se copia no habrá excepción.

EXCEPCIONES. Evitar copia en captura catch

Otra cuestión es que cuando “capturamos la excepción” con catch, dentro del mismo se hace una copia:

```
try{
    int num = 0;
```

```
    throw num; //esto es una copia NECESARIA por que num morirá depues del try
} // catch(int ex); //esto es OTRA copia por lo que gastamos más memoria.
catch(int &ex); //no es copia ya que vamos a la memoria donde existe la excepción
```

Al capturar el catch por referencia, evitamos la copia. Muy importante cuando la excepción lanzada son objetos (clases) que ocupan mucho en memoria Y PUEDEN SER SLICING OBJECTS al copiarlo

```
catch(myClass &ex);
```

EXCEPCIONES. Varios catch para varios throw

Ya sabemos que si hay un throw, debe haber un catch. Pero puede haber varios.

```
try{
    if (condicion)
        throw 0;} //lanza un entero
catch (std::string ex){}
catch (int ex){}; //este será el recogido ya que el throw es un entero.
```

si hubiera un catch, pero no del mismo tipo lanzado, el programa terminaría llamando a std::terminate().

Esto es para que si se lanzan varios throw en el código, cada catch tome el tipo correspondiente.

Si tenemos dos catch con tipos iguales, pillará el primero

```
try {
    throw 42;}
catch (int ex) {std::cout << "primero\n";}
catch (int ex) {std::cout << "segundo\n";} //no alcanzará este
```

Si queremos capturar de cualquier tipo que se lance, ponemos catch(…) que es válido en C++98. Si hay más catch, debería ser el último en ser puesto por que si no anula los otros.

```
try {
    throw 42;}
catch (std::string ex) {}
catch (...) {} //es el que se ejecutará.
```

EXCEPCIONES. clases herencias

Se puede lanzar de objetos heredados, pero los catch han de ser en orden inverso, por que si se pone clases origen antes, estas son más generales y pillarían toda excepción al ser las derivadas parte de las generales. Por eso hay que ir de mas especificas a padres:

```
try{do_something();}
catch(Nieta &ex){};
catch(Hija &ex){};
catch(Padre &ex{}; //si pusiera esta la primera, todo error de hijos quedaria anulado por esta
```

Pero esto se haría asi, si no usaramos polimorfismo (y estamos usando una función especifica que se llama de la misma manera y anula a las generales, que actua de diferente manera por cada clase heredada).
Si lo hacemos con virtual, ya solo hace falta poner un catch.

```
class Padre{
    Padre(const std::string &s) : _msg(s){} //constructor de copia necesario para throw
    virtual ~Padre(){};
    virtual const std::string funcion() const {return _msg;}
protected:
    std::string _msg;};
class Hija : public Padre{
public:
    Hija(const std::string &s) : Padre(s){};
    const std::string funcion() const override{return _msg + “Hija”} //override c++11
class Nieta : public Hija{
public :
    Nieta (const std::string &s) : Hija(s){}
    const std::string funcion() const override{return _msg + “Nieta”}  //override c++11
};
void do_something(size_t i){
    if(i == 2){throw Hija(“i es 2”)};
    if (i == 3{throw Nieta(“i es 3”)};}
int main(){
    for (size_t i = 0; i < 5; i++){
        try{
            do_something(i);}
        catch((Padre &ex){ //solo hace falta uno al ser polimorficos. ha de ser un tipo Padre más general
            std::cout << “Error pillado en : ” << ex.funcion() << std::endl;}}}
```

EXCEPCIONES. Relanzar

Se puede hacer un relanzando de una excepción y la sintaxis es con throw a secas.

```
try{
    for (size_t i{}, I < 5, ++i){ //c++11 por inicializacion int i{3}
        try{
            do_something();} //do_something hará un throw dentro.
    catch(...){ // (…) se meterá para cualquier throw lanzado dentro de do_something()
        throw; // si fuera throw tipo haría una copia y en clases haría object slicing.
            //dicho throw hara que nos sagamos del loop for, y pasaremos al siguiente catch.
    }
    catch(...){loquesea();}
}
```

EXCEPCIONES. Excepciones bloqueadas a solo la función.

Como hemos visto, se puede lanzar una excepción en una función y pasará a fuera de la función para ser recogida fuera, si es preciso. PERO podemos controlar que una excepción no pueda salir fuera de dicha función interna.

1. noexcept que es para C++11, es como una etiqueta aplicada a la función para que no salga de ahí.

```
void do_something(size_t I) noexcept { //se puede poner noexcept(true) o noexcept(false) para
        permitirlo o no.
    try{
        throw1;}
    catch(int ex){ throw;} //ese relanzado throw terminará el programa por std::terminate(), ya que no
            puede salir de la función al no haber más catch dentro de la misma.
}
```

2. throw() dentro de la función para C++98, pero ya está obsoleto.

```
void do_something(size_t I) throw() { //si se compila en posterior a C++98 daría advertencia
    try{
        throw1;}
    catch(int ex){ throw;} //ese relanzado throw terminará el programa por std::terminate(), ya que no
            puede salir de la función al no haber más catch dentro de la misma.
}
```

en el código anterior si hago esto:

```
void do_something(size_t I) throw(int){ //ahí si que podira relanzarlo ya que esto permite un int.
```

EXCEPCIONES. Excepciones en un Destructor.

Se puede pasar una excepción por un destructor de clase, pero solo es recomendable si no hay otras excepciones en curso:

```
class Padre{
public:
    Padre(){}
    ~Padre() noexcept(false){ //solo a partir de C++11
        throw 0;}
int main(){
    try{ Padre padre;
    }catch (int ex){…} //aqui llegaría si hemos metido el noexcept(false) dentro del destructor
    return 0;}
```

Si fuera sin noexcept(false) se considera automaticamente como noexcept(true) siempre en destructores, por lo que esto sería válido:

```
~Padre() { //valido para C++98
    try {
        throw 0;
    } catch (int ex) {
        // excepción atrapada dentro del destructor
    }
}
```

Para permitir sacar una excepcion en concreto en C++98 tendría que poner:

```
class Padre{
public:
    Padre(){}
    ~Padre() throw(int){ //solo valido en C++98
        throw 0;}
```

EXCEPCIONES. Standard

Ya hay una librería para usar excepciones ya construidas. Es con #include <exception>

Se puede ver más aquí: https://en.cppreference.com/w/cpp/error/exception.html

Dentro del link se pueden ver cuales son de C++98 (sin etiqueta) y cuales no.

Ej: Para este ejemplo vamos a considerar dos objectos polimórficos:

```
class Padre{…};
class Hija : public Padre{…};
```

Si hacemos esto en el main:

```
int main(){
    Padre padre1;
    Hija *p_hija = dynamic_cast<Hija*>(&padre1); //el puntero será nullptr o NULL en C++98 por
        que no se puede sacar info de Hija si el objecto creado es anterior (Padre)
    if (p_hija) {do_something(p_hija);}}
```

No pasa nada, por que al ser punteros podemos comprobar si no es nullptr y no entrar en segfault dentro de do_something(Hija *p_hija)

PERO si en vez de usar punteros usamos referencias:

```
int main(){
    Padre padre1;
    Hija &ref_hija = dynamic_cast<Hija&>(padre1); //las referencias no pueden tomar valor NULL
    if (ref_hija) {do_something(p_hija);}} //no se puede hacer
```

Cuando se hace por referencia, si no se puede realizar el programa lanzará una excepción standard que se puede recoger, por lo tanto en vez del if, debemos usar:

```
int main(){
    try{
        Padre padre1;
        Hija &ref_hija = dynamic_cast<Hija&>(padre1);
        do_something(ref_hija);
    } catch(std::exception &ex){ //más especifico sería catch(const std::bad_cast &ex)
        //notar qeu es &ex ya que al ser referencia permitiría el polimorfismo ya que std::exception ex, haría
        //object slicing
        std::cout << “Error : ” << ex.what() << std::endl;} //lanzará un std::bad_cast.
```

Eso para ver el error, si quiero imprimirlo bajo catch(std::exception) puedo hacer:

```
catch(std::exception &ex){
    if(ex) == typeid(std::bad_cast))
        std::cout << “Se lanzo un bad_cast: ” << ex.what();
    else
        std::cout << “Otra excepcion : ” << ex.what();}
```

EXCEPCIONES. Lanzar excepciones Standard

Hemos visto que cuando se autolanzan, no se usa el throw, pero podemos forzar lanzar una excepción standard.

Si tenemos una clase Estudiante que almacenamos un número determinado en un array:

```
class Estudiante{
```

```
private:
    std::string _alumno[5];}
```

Si luego tenemos un getter para sacar info de los estudiantes almacenados, podemos controlar el número pasado de índice:

```
const std::string get_estudiante(size_t index){
    const std::string msg = “Indice fuera de rango”;
    if (index >= 5) //al ser size_t no puede ser negativo
        throw std::out_of_range(msg); //se puede mandar msg como dice aqui:
            https://en.cppreference.com/w/cpp/error/out_of_range.html
    return (_alumno[index]);}
```

EXCEPCIONES. Excepciones derivadas de la Standard

std::exception es una clase que admite el polimorfismo de what(), por lo tanto podemos añadir otras excepciones:

```
class DividirPorCeroExcepcion : public std::exception{ //la hacemos heredera de standard
public:
    const char *what() const noexcept override { //hay que redefinirla por que es virtual,  y si    no, no se podrá generar un objeto de esta clase, siendo abstracta.
        return (“Dividido por cero detectado”);}};
int dividir(int a, int b){
    if(b == 0)
        throw DividirPorCeroExcepcion(a, b);
    return (a/b);
}
int main(){
    try{
        dividir(10,0);
    } catch (std::exception &ex){
        std::cout << ex.what(); //como es polimorfica mostrará la de la clase creada derivada de la standard
    }}
}
```

CASTING

C type casting. Implicito y explicito

En C++ se puede hacer un C-type casting de tal manera que podemos hacer un casteo implícito:

```
double num1 = 5.2;

int num2 = num1; //casting implicito ya que combierte en tiempo de compilacion a int el num1
```

O explicito:

```
double num1 = 5.2;

num1 = (int)(num1 + 3.1); //será 8 en vez de 8.3
```

Static Casting

Pero en C++ tiene su propia forma de hacerlo, que en realidad no puede hacer mucho más de lo que lo hace con el método c-type casting, pero facilita la lectura, y aparte introduce mejoras de seguridad. El casteo explicito anterior podría haber sido escrito con static casting como:

```
double num1 = 5.2;

num1 = stating_cast<int>(num1 + 3.1); //será 8
```

y como el c-type casting, este casteo será procesado en tiempos de compilación.

Dynamic_cast

Dynamic Casting se produce en tiempo de ejecución, por lo que es más “pesado” y peor rendimiento que static_casting, pero nos ofrece una seguridad extra. Se usa en objetos polimórficos:

```
class Entidad{
public:
    virtual void Funcion();
};
class Jugador{
};
class Enemigo{
};
int main(void){
    Jugador *jugador1 = new Jugador();
    Entidad *entidad1 = (Entidad*)jugador1; //sin problema compilacion.
    Entidad *entidad2 = static_cast<Entidad*>(jugador1);//misma direccion memoria que entidad1
    Entidad *entidad2 = dynamic_cast<Entidad*>(jugador1);//misma memoria que entidad1
}
```

En el ejemplo arriba, creamos un objeto jugador1 que pillará una dirección de memoria como objecto Jugador. Como hereda de Entidad, jugador1 será también una Entidad, y por lo tanto se puede hacer el casteo en entidad1 2 y 3. Los tres tendrán la misma dirección de memoria. PERO!!!

```
int main(void){
    Jugador *jugador1 = new Jugador();
    Enemigo *enemigo1 = (Enemigo*)jugador1; //Compila, pero INDETERMINADO
    Enemigo *enemigo2 = static_cast<Enemigo*>(jugador1);//Error. Conversion inválida más seguro que ctype casting!!
    Enemigo *enemigo3 = dynamic_cast<Enemigo*>(jugador1);//será nulptr
}
```

Aquí está el problema. Un Enemigo es una Entidad, y un Jugador es una Entidad. PERO un Jugador no es un Enemigo y viceversa. Por lo tanto:

1. enemigo1 compilará por que en tiempo de compilación no se sabe lo que es, pero al ejecutar, hará cosas indefinidas. jugador1 NO es un Enemigo y si queremos acceder a una función del Enemigo, no podrá.

2. enemigo2 no dejará compilar por que ya de por si reconoce que los dos tipos no coinciden asi que protege, mientras el caso 1 que era un ctype casting dejaba compilar.

3. ememigo3 compilará, pero nos dará que el puntero es NULL, por que no puede sacar la información. Lo hará en tiempo de ejecución y por eso tarda más, pero si luego en nuestro main, ponemos algo como:

```
if (enemigo3) bla bla bla
```

Estaremos protegidos y no habrá segfault. De tal forma que preguntando podriamos ver si enemigo3 es un enemigo o no. En este caso no, sería un jugador y por eso da NULL.

Si hicieramos:

```
Jugador *enemigo4 = dynamic_cast<Jugador*>(jugador1);
```

Sí que tendría una dirección válida enemigo4 ya que enemigo4 sí que sería un jugador.

Reinterpret_cast

Sería el homologo al ctype cast, de tal manera que “deja” hacer lo que el programador quiera, pero que luego puede dar muchos problemas, o sea que es muy peligroso.

reinterpret_cast, solo permite hacer casteo a PUNTEROS o REFERENCIAS. Por que lo que trata de hacer es asignar la direccion de variable y cambiarla

```
float pi = 3.14f;
std::cout << (int)pi; //funciona y seria 3.
std::cout << *reinterpret_cast<int*>(&pi); //funciona pero daría un valor erroneo ya que tranfiere la dirección de memoria float a int, es decir interpreta como un int, lo que era un float. Lo dereferenciamos.
```

```
std::cout << reinterpret_cast<int>(pi); //no compilaria. Son tipos diferentes.
```

En este siguiente caso las tres sería la misma dirección de memoria, pero si dereferenciamos veremos que el valor es diferente, por que se interpreta uno como int, y el otro como float, aunque sea la misma dirección de memoria.

```
std::cout << &(pi);
std::cout << reinterpret_cast<int*>(&pi); //no dereferenciado. Imprime la direccion de memoria.
std::cout << reinterpret_cast<float*>(&pi);
```

Entonces, donde se usaría, siendo tan peligroso? Imaginemos un juego con datos de jugador:

```
struct GameState{
    int     level;
    int     health;
    int     points;
    bool  GameCompleted;
    bool  BossDefeated;
};
int main(){
    GameState gs = {66, 100, 999, false, false};
    char kk[sizeof(GameState)]; //memoria en el stack del mismo tamaño de GameState. 14bytes.
    std::memcpy(kk, &gs, sizeof(GameState)); //copia los bytes de gs al array kk.
    std::cout << *(kk) << std::endl; //imprime la letra B, por que B en ASCII es 66.
    std::cout << *((int*)kk) << std::endl; //imprime 66

    std::cout << *reinterpret_cast<int*>(kk) << std::endl; //imprime 66
    std::cout << *reinterpret_cast<int*>(kk+4) << …; //imprime 100. +4k suma 4bytes a inicio d kk
    std::cout << *reinterpret_cast<int*>(kk+8) <<…; //imprime 999
    std::cout << *reinterpret_cast<bool*>(kk+12) <<…; //imprime 0 de false
    std::cout << *reinterpret_cast<bool*>(kk+13) <<…; //imprime 0 de false
}
```

Las lineas de reinterpret_cast, no funcionarian con static_cast, ni con dynamic_cast. Si que lo haría si fueran (int*)(kk+4) etc… pero es peligroso.

El otro “sitio” donde utilizar reinterpret_cast es con un void puntero:

```
void *p = &gs; //puntero generico, sobre ese gs que venia de la estructura GameState.
GameState *real = reinterpret_cast<GameState*>(p); //reinterpretación
```

Const_cast

Lo que hace es saltarse “a la torera” la etiqueta const, puesta sobre un objeto para poder modificarlo. Tienen que ser siempre punteros o referencias.:

```
class Demo{
public:
    int var = 1;
    void func() const{ //const no permite modificar el objeto
        var = 2; //error de compilación. No se puede alterar.
        (const_cast<Demo*>(this))→var = 2; //lo pasará. Tiene narices!! XDD
    }
};
```

O:

```
const Tipo* ptr = …;
Tipo* mod = const_cast<Tipo*>(ptr); //quita el const.
```

O referencias:

```
const Tipo& r = …;
Tipo& mod = const_cast<Tipo&>(r); //quita el const.
```

FUNCIONES TEMPLATE

Cuando tenemos funciones Overload, como esto:

```
int    f_max(int a, int b){
    return (a > b) ? a : b;}
double  f_max(double a, double b){
    return (a > b) ? a : b;}
```

Si tenemos repetición de lógica tendremos que repetir mucho. Aquí es donde entran los TEMPLATES (plantillas) de funciones:

```
template <typename T> T f_max(T a, T b){
    return (a > b) ? a : b;}
```

Para definirlo tiene que estar toda la linea junta. template <typename T> no es como un include donde le digo que voy a usar tipos variables. Si tuvieramos varias funciones que queremos usar Template TENEMOS que ponerles su palabra template en ellos. Esto NO es válido:

```
template <typename T> T f_max(T a, T b){
    return (a > b) ? a : b;}
T f_min(T a, T b){ return (a < b) ? a : b;}
```

Sino que sería así:

```
template <typename T> T f_max(T a, T b){
    return (a > b) ? a : b;}
template <typename T> T f_min(T a, T b){
    return (a < b) ? a : b;}
```

Para comprobar como funciona se puede ir a la web: https://cppinsights.io dándole al play con el código copiado el generará las funciones en tiempo de compilación.

En C++11, podriamos hacer luego:

```
auto variable = f_max(a, b); //auto NO C++98!!!!
```

Para que no de error en el compilador, el cuerpo de la función ha de poder hacer las operaciones que tenga con las variables que se les pase. De echo, una función multiplicación podria multiplicar enteros, doubles, floats, etc... pero no strings (que sí que podrían sumarse) asi que si hago un template multiplicando las variables, y luego paso un std::string, el compilador se quejaría.

```
template <typename T> T multiply(T a, T b){
    return a * b;} //ERROR COMPILACION por que no se puede multiplicar std::string
int main(){
    std::string a = "hola";
    std::string b = "mundo";
    multiply(a, b);
}
```

Pero también el tipo debe ser el mismo, por que si no dará error de compilación:

```
template <typename T> T multiply(T a, T b){
    return a * b;} //ERROR COMPILACION por que no se puede multiplicar std::string
int main(){
    double a = 4.3;
    int b = 3; //debe ser el mismo tipo
    multiply(a, b);}
```

PASAR POR REFERENCIA

Para cambiar los valores de unas variables dentro de una función que no devuelve nada y que se reflejen fuera sería así:

```
template <typename T>
void swap(T &a, T &b){
    T aux  = a;
    a      = b;
    b      = aux;}
```

VARIOS TIPOS DE VARIABLES EN EL MISMO TEMPLATE

Como hemos dicho, si pasamos dos tipos distintos a una que admite una sola T, el programa no compilaría, para hacerlo hay que hacer esto:

```
template <typename T, typename V, typename D> D funcion(T a, V b);
```

//aquí D es el tipo de devolución de la función. Ya puede ser void, int, double, etc… Puede ser cualquier letra.

PASAR FUNCIONES COMO PARÁMETROS

Pero es que podemos pasar cualquier cosa:

```
template <typename T, typename F, typename K> void funcion(T*, F (*f)(K));
```

//esto crearia una void funcion, que permitiría como parametros, un tipo puntero de cualquier tipo (podria ser una matriz, ya que apunta al primer elemento) y como segundo parametro una funcion (las funciones pasadas por parametro se convierten automaticamente a puntero. no necesitan &funcion, aunque también eso se puede hacer) que devolviera el parámetro F y que tomara como parametro único de cualquier tipo K.

Pero en el ejemplo anterior podemos ser mucho más genéricos:

```
template <typename T, typename F> void funcion(T*, F f)); //quitamos K
//aquí F es el tipo de variable de f (int f, double f, float f,…) pero también puede ser F el tipo de retorno de una funcion cualquier que se le pase, y f seria cualquier funcion (void funcion(int a, int b), std::string funcion(void), etc…) Incluso si se quiere hacer que los parametros sean const o no const (void funcion (const int num) o void funcion (int num). E incluso podria ser cualquier objeto.
```

ARGUMENTOS EXPLICITOS EN LLAMADAS A TEMPLATES

Podemos pasarle un argumento explícito a la hora de llamar a la función. Se hace pasándo el tipo entre < >:

```
template <typename T> T maximun(T a, T b){
    return (a > b) ? a : b;}
int main(){
    int         var1 = 32;
    double      var2 = 42.4;
    std::string var3 = “hola”;
    maximun(var1, var2); //daría error de compilación. Tendria que usar varios tipos en la misma función
    maximun<double>(var1, var2); //funciona. Haria conversión a double
    maximun<double>(var1, var3); //error de compilación. No se puede convertir string a double
}
```

TEMPLATE SPECIALIZATION

Hay variables que no tiene sentido pasarle a un mismo template y que no darían el resultado que esperamos. Así si pasamos a este template maximun:

```
template <typename T> T maximun(T& a, T& b){
return (a > b) ? a : b;}
int main(){
    const char* var1 = “hola”;
    const char* var2 = “adios”;
    std::cout << “el maximo es: “ << maximun(var1, var2); //no devolverá lo esperado
```

No devolverá “hola” por que lo que devolverá son const char* y eso son punteros. Es decir comparará punteros. Entonces tenemos que hacer una “especialización” específica del template para ciertos tipos de variables:

```
template <typename T> T maximun(T& a, T& b){
return (a > b) ? a : b;}
template <>
const char* maximun(const char* a, const char* b){ // podriamos pasar por referencia el puntero: const char* const& a, pero copiar un puntero es muy ligero.
    return (std::strcmp(a, b) > 0 ? a : b); //strcmp retorna -1 si es menor, 0 iguales o >0 si el primer parametro es menor}
int main(){
    std::cout << “el maximo es: “ << maximun(var1, var2); //ahí si que lo devolverá bien}
```

Esta declaración de la especialización no tiene que venir justo debajo de la generalista. Sí tiene que venir después, pero lo que le hace funcionar es que coincide el nombre de la función para que el compilador la reconozca (aparte del template <>)

PERO, esto también funcionaría si se añadiera sin la expecialización:

```
const char* maximun(const char* a, const char* b){
    return (std::strcmp(a, b) > 0 ? a : b); }
```

Es decir, una funcion overloada normal sin especialización (template <>), por lo que es una cuestión de “completar” la plantilla (con template <>) o no, para hacer referencia que es una plantilla completa. Vamos.. por claridad y arquitectura.