Informática Industrial

Estos apuntes se han realizado tomando en cuenta los apuntes de la asignatura y el contenido dado en clase por el profesor Miguel Hernando Gutierrez.

Índice

0. Sesión 0. Introducción a la POO y repaso de C.

0.1. Repaso de C.

0.1.1. Variables y tipos de datos.

Tipos de variables:

- Automáticas (`int, float, double,...`). Se caracterizan por ser gestionadas automáticamente por el compilador y se mantienen mientras esté activo el bucle `main`. Ejemplo concreto sobre el uso de `float`:

```c
#include <stdio.h>

int main() {
    int a = 3;
    int b = 2;
    double resultado1 = 3.0 / 2;          /*(1)*/
    int resultado2 = 3 / 2;               /*(2)*/
    printf("3.0 / 2 = %.1f\n", resultado1);
    printf("3 / 2 = %d\n", resultado2);   /*(3)*/
    return 0;
}
```

  Comentarios a realizar:

  - Línea (1). Se realiza una división como números `float`.
  - Línea (2). Se realiza una división como números `int`.
  - Línea (3). En conclusión, se observa que no es lo mismo dividir un `int` o un `float`.

- Estáticas (`static`). Se diferencian de las automáticas porque no son dinámicas, no son gestionadas por el compilador, permanecen indefinidamente. Ejemplo:

```c
#include <stdio.h>
int foo() {
    static int var_static = 7;
    var_static++;
    return var_static;
}
int main() {
    printf("%d\n", foo());
    printf("%d\n", foo());
    printf("%d\n", foo());
    printf("%d\n", foo());
    return 0;
}
```

  Salida por consola:

```
8,9,10,11
```

Tipos de datos de usuarios:

- `struct`: colección de datos. Ejemplo:

```
struct punto_t{
    p.x;
    p.y;
};
```

- `enum`: colección de enumerados, constantes enteras con nombre. Es decir, es un tipo que contiene un conjunto de símbolos asignados para distintos valores enteros. Ejemplo:

```
enum_palo_naipe{OROS,BASTOS,COPAS,ESPADAS};          /*(1)*/
enum_pablo_naipe{OROS = 0, BASTOS, COPAS, ESPADAS}; /*(2)*/
```

  Línea (1). Queremos decir que existen 4 tipos de constantes enteras y cada una tiene un valor.

  Línea (2). En este caso, `OROS` es 0 y el resto de palos valen 1.

Consideraciones:

- En C, las variables se declaran al inicio de los bloques de instrucciones `{ }`.
- Las palabras clave `struct` y `enum` se escriben también en las declaraciones de variables.

Palabras clave:

- `typedef`: palabra clave para "alias" de tipos de variables.

```
typedef struct punto_t {
    double x;
    double y;
} punto_t;
```

- `sizeof`: palabra clave que devuelve el tamaño del argumento en **bytes**.

```
typedef enum pieza_t { REINA = 0, REY } pieza_t;
printf("%d", sizeof(pieza_t));
```

0.1.2. Funciones.

Paso de datos por **valor**. En este caso, el valor del parámetro **no** podrá ser modificado.

```
void print_carta(naipe_t n);
```

Paso de datos por **referencia**. Sí se puede modificar el valor del parámetro.

```
void limpiar_carta(naipe_t *n); //tiene sentido, solamente
   puedes limpiar una carta si tienes acceso a su valor, luego
   es paso por referencia.
```

Ejemplo:

```
#include <stdio.h>
#define SIZE_OF(x) printf("Size of " #x " --> %zu\n",
   sizeof(x)) /*(1)*/

int main(void) {
    SIZE_OF(char);
    SIZE_OF(int);
    SIZE_OF(long int);
    SIZE_OF(short int);
    SIZE_OF(long double);
    SIZE_OF(int *);
    SIZE_OF(long int*);
    return 0;
}
```

Línea (1):

- Se define una macro llamada `SIZE_OF` que toma un argumento `x`.
- En `#x`, el símbolo `#` convierte el argumento `x` en una cadena literal.
- `sizeof(x)` calcula el tamaño en bytes del tipo de dato o variable `x`.
- `%zu` es el especificador de formato adecuado para imprimir `size_t`, que es el tipo devuelto por `sizeof`.

0.1.3. Punteros.

Un puntero es una variables que contiene **la dirección de memoria** de un dato u otra variable. Operadores (unarios) relacionados:

- `*` : operador indirección, devuelve el valor apuntado por el operando. Es decir, accedemos al contenido.
- `&` : operador dirección, devuelve la dirección de memoria del operando.
- `=` : operador asignación, asigna una dirección de memoria a un puntero o un valor a la dirección apuntada.
- `->` : operador flecha, accede a miembros de una estructura a través de un puntero.

Ejemplo 1:

```
#include <stdio.h>

int main() {
    int var1 = 10, var2 = 20;
    int* puntero;                /*(1)*/
    int *puntero2;               /*(2)*/
    puntero = &var1;             /*(3)*/
    printf("El valor del puntero es %d\n", *puntero); /*(4)*/
    *puntero = 12;               /*(5)*/
    printf("El valor ahora de var1 es %d que es igual al valor
       del puntero %d\n", var1, *puntero); /*(6)*/
    puntero2 = &var2;            /*(7)*/
    printf("la suma de ambas variables es %d", *puntero +
       *puntero2);
}
```

Comentarios a realizar:

- Línea (1). Defino un puntero a entero llamado `puntero`.
- Línea (2). Defino otro puntero a entero llamado `puntero2`, la posición del asterisco es indiferente.
- Línea (3). Asigno a `puntero` la dirección de la variable `var1`.
- Línea (4). Imprimo el contenido de la dirección a la que apunta `puntero`, es decir, el valor de `var1`.
- Línea (5). Modifico el valor de la variable apuntada por `puntero`, ahora `var1` vale 12.
- Línea (6). Imprimo el nuevo valor de `var1` y verifico que coincide con el valor apuntado por `puntero`.
- Línea (7). Asigno a `puntero2` la dirección de la variable `var2`.

Ejemplo 2:

```
struct Persona {
    char nombre[20];
    int edad;
};                                                  /*(1)*/

struct Persona p = {"Ana", 30};                     /*(2)*/
struct Persona *ptr = &p;                           /*(3)*/
printf("%s tiene %d años", ptr->nombre, ptr->edad); /*(4)*/
```

Comentarios a realizar:

- Línea (1). Se define una estructura llamada `Persona` que contiene un nombre (cadena de caracteres) y una edad (entero).
- Línea (2). Se declara una variable `p` de tipo `struct Persona` e inicializa con el nombre "Ana" y edad 30.
- Línea (3). Se declara un puntero a estructura `ptr` y se le asigna la dirección de `p`.
- Línea (4). Se imprime el contenido apuntado por `ptr`, accediendo a los campos mediante el operador `->`. El resultado es: `Ana tiene 30 años`.

Ejemplo 3:

```
double Punto::* pr_double;      /*(1)*/

pr_double = &Punto::y;          /*(2)*/

p1.*pr_double = 3.0;            /*(3)*/

punt_p ->* pr_double = 4.0;     /*(4)*/
```

Comentarios a realizar:

- Línea (1). Se declara un puntero a miembro de tipo `double` perteneciente a la clase `Punto`.
- Línea (2). Se asigna al puntero `pr_double` la dirección del miembro `y` de la clase `Punto`.
- Línea (3). Se accede al miembro apuntado por `pr_double` en el objeto `p1` y se le asigna el valor 3.0 mediante el operador `.*`.
- Línea (4). Se accede al miembro apuntado por `pr_double` usando un puntero a objeto `punt_p` con el operador `->*`, y se le asigna el valor 4.0.

0.1.4. Punteros. Aritmética

Operadores (unarios) relacionados:

- `++` : incrementa una posición relativa al tamaño del objeto apuntado.
- `--` : decrementa una posición relativa al tamaño del objeto apuntado.

Ejemplo a modo de recordatorio:

```
i++;//incremento 1 unidad y luego se lo asignamos el valor a i.
++i;//primero asignamos el valor y luego incrementamos el valor.
```

Ejemplo:

```
#include <stdio.h>

int main() {
    int coleccion[4] = {10, 20, 30, 40};  // Inicializa el
       arreglo con 4 elementos
    int* p = coleccion; // El puntero p señala al primer
       elemento de la colección

    // Imprime los valores usando el puntero sin desplazamiento
    printf("Valor del primer elemento: %d\n", *p);

    p++; // Avanzando al siguiente elemento
    printf("Valor del segundo elemento: %d\n", *p);

    p++; // Avanzando al tercer elemento
    printf("Valor del tercer elemento: %d\n", *p);

    p++; // Avanzando al cuarto elemento
    printf("Valor del cuarto elemento: %d\n", *p);

    // Mostrando la diferencia de direcciones de memoria
    printf("\nDirección del primer elemento: %p\n", coleccion);
    printf("Dirección del cuarto elemento: %p\n", p);
    printf("Diferencia en posiciones: %ld\n", p - coleccion);

    // Usando notación de arreglo con punteros
    p = coleccion; // Regresa p al inicio del arreglo
    printf("\nUsando notación de arreglo con punteros:\n");
    for(int i = 0; i < 4; i++) {
        printf("coleccion[%d] = %d, usando puntero: %d\n", i,
           coleccion[i], *(p + i));
    }

    return 0;
}
```

Cuestión: ¿Son correctas todas las instrucciones?

```
int lista[3] = {1,2,3};

int* p = lista;

printf("%d", *(p+2));
printf("%d", *(lista + 2));
printf("%d", *(p++) );
printf("%d", *(lista++));
```

```
                                                puntero constante
printf("%d", *(lista++)); //no compila porque lista es un
printf("%d", *(p++) ); //correcta, imprime "1"
printf("%d", *(lista + 2)); //correcta, imprime "3"
                                          lectura, imprime '3'
printf("%d", *(p+2)); //correcta por ser una operación de

int* p = lista;

int lista[3] = {1,2,3};
```

0.1.5. Constantes, `define` and `const`.

Existen dos formas principales para definir constantes.

- `define`: es una directiva de preprocesador, no una función ni una variable. Se utiliza para constantes simples o macros. No tiene tipo de dato y no ocupa memoria. Ejemplo:

```
#define PI 3.14159
#define MAX_SIZE 100
```

  Ejemplo de macros:

```
#include <stdio.h>
#define CUADRADO(x) ((x) * (x))
int main() {
    int resultado = CUADRADO(5); // Se expande a: ((5) *
       (5))
    printf("%d", resultado); //25
    return 0;
}
```

- `const`: declara una variable constante con un tipo de dato específico. Se almacena en memoria y respeta las reglas de alcance (scope). Es un concepto que usaremos ampliamente en nuestra programación en C++. Esto es porque es especialmente útil para:
  - Por motivos de autoproteger tu código.
  - Porque es la manera que una función pueda recibir literales y/o variables.

  Ejemplo:

```
const double PI = 3.14159;
const int MAX_SIZE = 100;
```

  Cabe destacar que se puede escribir la palabra `const` antes o después del tipo básico sin cambiar el significado. Ejemplo:

```
//Se define una constante de tipo real con un valor:
const double pi = 3,14159; //equivalente a
double const pi = 3,14159; //forma recomendada

//Se puede usar con auto:
const auto pi = 3,14159; //equivalente a
auto const pi = 3,14159;
```

  Además, se puede compartir la línea de declaración si comparte el tipo de datos básico. Ejemplo:

```
int * const a = &b, *const d = &b, *e, m, p = 3, &q = p;
```

Cuestión: ¿Cuáles son las diferencias entre estas dos instrucciones?

```
const int *p1;
int* const p2;
```

Solución: el truco es leer de derecha a izquierda.

- La primera línea es una declaración de un puntero `p1` que apunta a un entero constante. Con lo cual, con este puntero no podré modificar el contenido de ese dato constante. Es un puntero de lectura.
- La segunda línea es un puntero constante `p2` que apunta a un entero. Es decir, el puntero apunta siempre al mismo sitio. Sí se puede modificar el valor apuntado, pero no puede cambiar de lugar. ¿Cuál es una posible aplicación?

```
int coleccion [4] = {1, 2, 3, 4}; // El puntero coleccion
   es un puntero constante que apunta al primer elemento
```

Cuestión: ¿Cuál es el significado de las siguientes instrucciones?

```
const int * const p = &b;
int const * const p = &b; //forma recomendada
```

Respuesta: ambas son equivalentes y su significado es declarar un puntero constante a un entero constante.

Cuestión: ¿Son correctas todas las instrucciones?

```
int var1 = 10, var2 = 20;
const int * p1 = &var1;              /*(1)*/
int * const p2 = &var2;              /*(2)*/
{
    *p1 = 20;                        /*(3)*/
    p1 = &var2;                      /*(4)*/
    *p2 = 20;                        /*(5)*/
    p2 = &var1;                      /*(6)*/
}
```

Comentarios a realizar:

- Línea (1). Se declara un puntero `p1` a entero constante: no se puede modificar el valor apuntado, pero sí cambiar a qué dirección apunta.
- Línea (2). Se declara un puntero constante `p2` a entero: no se puede cambiar la dirección a la que apunta, pero sí modificar el valor apuntado.
- Línea (3). Error: intenta modificar el valor apuntado por `p1`, pero al ser un puntero a constante, esto no está permitido.
- Línea (4). Correcto: `p1` puede cambiar la dirección a la que apunta.
- Línea (5). Correcto: se modifica el valor apuntado por `p2`, lo cual está permitido porque el valor no es constante.
- Línea (6). Error: se intenta cambiar la dirección a la que apunta `p2`, pero es un puntero constante.

0.1.6. Arrays

Un array es una variable estructurada, en el que cada elemento se almacen de forma consecutiva en memoria.

```
int coleccion [4] = {1, 2, 3, 4}; //coleccion es puntero
   constante a primer elemento
for (int i = 0; i < 4; i++){
    printf("%d", coleccion[i]);
}
```

0.1.7. Cadenas de caracteres.

Las cadenas de caracteres son arrays de caracteres (`char`) que terminan siempre con el carácter especial "\0". Ejemplo:

```
char cadena[] = "Hola";
char cadena2[] = { 'H', 'o', 'l', 'a', '\0' };
printf("%d %d", sizeof(cadena), sizeof(cadena2));
```

Salida por consola:

```
5 5
```

Se emplea la librería:

```
#include<string.h> /*en C++ es*/ #include<cstring.h>
```

Algunos servicios típicos:

- `strlen(<cadena>)`: Devuelve la longitud de la cadena sin contar el carácter "\0".
- `strcpy(<cadena destino>, <cadena origen>)`: Copia la cadena de origen a destino.
- `strcmp(<cadena1>, <cadena2>)`: Compara las dos cadenas lexicográficamente. Si las cadenas son iguales devuelve 0, un número negativo si la `cadena1` precede alfabéticamente a la `cadena2` y un número positivo si la `cadena1` es posterior alfabéticamente a la `cadena2`.
- `strncpy(<cadena destino>, <cadena origen>, <n>)`: Copia hasta `n` caracteres de la cadena de origen a destino, útil para evitar desbordamientos.
- `strncmp(<cadena1>, <cadena2>, <n>)`: Compara lexicográficamente hasta `n` caracteres de dos cadenas.
- `strcat(<cadena destino>, <cadena origen>)`: Concatena (añade) la cadena de origen al final de la cadena destino.

Cuestión: recorra una cadena e imprima cada uno de sus caracteres en pantalla.

```c
#include <stdio.h>
#include <string.h>

int main(){

    char cadena[] = "Hola";
    char* pc = NULL;
    int i;
    //A - []
    for (i = 0; i < strlen(cadena); i++) {
        printf("%c", cadena[i]);
    }
    puts("");

    //B - puntero explícito
    pc = cadena;
    for (; *pc != '\0'; pc++) {
        printf("%c", *pc);
    }
    puts("");

    //C -
    printf("%s", cadena);
}
```

Salida por consola:

```
Hola
Hola
Hola
```

0.1.8. Estructuras de control.

- Secuenciales.

```
{ //Bloque de instrucciones
    instrucción 1;
    instrucción 2;
    instrucción N;
}
```

- Condicionales.

  - if

```
if (condicion)
    instruccion1;
if (condicion){
    instruccion 1;
    instruccion 2;
}
```

  - if/else

```
if (condicion)
    instruccion1;
else
    instruccion2;
if (condicion){
    instruccion 1;
    instruccion 2;
}
else{
    instruccion 3;
    instruccion 4;
}
```

  - if/else anidado

```
if(condicion1)
    instruccion1;
else if(condicion2)
    instruccion2;
else if(condicion3)
    instruccion3;
else if(condicion4)
    instruccion4;
else
    instruccion5;

instrucción 6;
instrucción 7;
```

- switch

```
switch (expresion_eval_como_cte){
    case constante1:
        instrucciones;
        break;
    case constante2:
        instrucciones;
        break;
    /*... */
    default:
        instrucciones;
}
```

- Iterativas.
  - while

```
while (condicion){
    instrucción 1;
    instrucción N;
}
```

  - do-while. Se ejecuta siempre al menos 1 vez.

```
do {
    instrucción 1;
    instrucción N;

} while (condicion);
```

  - for

```
for (inicialización; condición; incremento/decremento){
    instrucción 1;
    instrucción N;
}
```

Ejemplo 1: bucle anidado for

```
void foo(int n, int vector[]){
    int i, j, temp;
    for(i=0; i < n-1; i++){
        for(j=i+1; j < n; j++){
            if(vector[i] > vector[j]){
                temp = vector[i];
                vector[i] = vector[j];
                vector[j] = temp;
            }
        }/* j */
    }/* i */
}
```

Ejemplo 2: bucle anidado while, for

```
typedef char BOOL;    /* no hay tipo BOOL en C */

void main(){// Algoritmo de números perfectos
    int numero, cont, suma;
    BOOL encontrado;
    encontrado = 0; numero = 101;
    while (!encontrado){
        suma = 1;
        for (cont = 2; cont < numero; cont++){
            if (numero % cont == 0)
                suma += cont;
        }
        if (suma == numero){
            encontrado = 1;
        }else numero++;
    }
    printf("El numero es = %d\n", numero);
}
```

- Comentario importante: todo en C y C++ tiene valor y las sentencias de control, utilizan estos valores en sus condiciones. Ejemplo:

```
#include <stdio.h>
int main(){
    int a  = 3;
    printf("%d", a == 3); // 1
    return 0;
}
```

0.1.9. Ejemplos de clase.

Se pide un programa para obtener las cifras de un número introducido por el usuario.

```
#include <stdio.h>

int main(){
    int numero, i = 0;
    int digitos[10];
    printf("introduce numero:");
    scanf("%d",&numero);
    printf("\n los dígitos del numero %d son:\n", numero);
    while (numero){
        digitos[i++] = numero % 10;
        numero /= 10;
    }
    while(i)printf("%d\n", digitos[--i]);
}
```

Alternativa usando logaritmos:

```c
#include <stdio.h>

int main(){
    int numero, base = 10;
    printf("introduce un numero:");
    scanf("%d", &numero);
    printf("\n Los dígitos del numero %d son:\n", numero);
    while (numero / base) base *= 10;
        base /= 10;
    while (base){
        printf("%d\n", (numero / base) % 10);
        base /= 10;
    }
}
```

Utilizando una estructuración correcta, podemos definir un proyecto tal que:

```c
#include <stdio.h>

void imprime_digitos(int n);

int main(){
    int numero;
    printf("introduce un numero:");
    if(scanf("%d", &numero)!= 1) return -1;
        printf("\nLos dígitos del numero %d son:\n", numero);
    imprime_digitos(numero);
}

void imprime_digitos(int n){
    int base = 10;
    while (n / base) base *= 10;
        base /= 10;
    while (base){
        printf("%d\n", (n / base) % 10);
        base /= 10;
    }
}
```

0.2. Introducción a la POO.

La Programación Orientada a Objetos nace porque al realizar programas grandes, es necesaria seguir una estructura organizada. La POO es útil para generar código reutilizable y se puede hacer código más eficiente desde el punto de vista del programador. Esta eficiencia significa en este contexto que el código es un lenguaje cercano al de la máquina.

Finalidad de la POO:

1. Generar código reutilizable.
2. Hacer un código eficiente.

En la POO se hacen pactos entre los datos y las funciones. Ejemplo, si realizamos un ajedrez, el centro de atención es el objeto, definir un tablero, las piezas, etc.

0.2.1. Elementos básicos de la POO.

1. Objeto. Es una entidad que tiene unos atributos (=datos) y unas formas de operar sobre ellos (métodos = funciones).
    - Métodos definen el comportamiento.
    - Los atributos definen el estado.
2. Clase. Es la definición de un tipo de objeto. Es el universal de objeto. Se dice que un objeto determinado (mi perro Jimmy) es una **instancia** de una clase Perro.
3. Métodos. Función implementada dentro de una clase para realizar operaciones con un objeto de la misma. Los métodos son comunes a la clase, pero trabajan con los datos específicos de cada trabajo.
4. Atributos. Datos contenidos en un objeto. Se declaran en la clase pero se instancian en cada objeto. Son específicos de cada objeto.
5. Mensaje. Acción de pedir la ejecución de un método. Remarca el concepto de responsabilidad. Es el objeto el responsable de la acción.

0.2.2. Características principales de la POO.

Tradicionalmente se han considerado 4 características principales:

1. Abstracción. Permite agrupar identidades para almacenarlas en clases.
2. Encapsulamiento. Es el concepto de que en un objeto, los datos y funciones deben de ir juntos encapsulados. Cada objeto debe ser tratado como una caja negra. En esa caja negra hay variables de estado y operaciones internas que no veo ni sé cómo funcionan. Interacciono con el objeto por medio de su carcasa o interfaz. Lo que puedo ver y usar del objeto es la interfaz.

3. Herencia. Mecanismo que nos permite crear clases derivadas (especialización) a partir de clases base (generalización). Librería de clases: conjunto de clase interconectadas.

4. Polimorfismo. Capacidad de un objeto de saber cómo realizar una acción que siendo genérica a más clases de objetos, tiene un modo específico de realizarse. Quiere decir que podemos crear funciones básicas en las clases básicas que luego pueden ser redefinidas en un objeto concreto.

5. Control en la construcción y destrucción de objetos. Los objetos por el hecho de ser creados se inicializan controladamente, y al ser destruidos, se eliminan controladamente.

0.2.3. Ejemplo. Centroide de una nube de puntos.

En el la programación estructurada clásica, los datos son un elemento externo al programa y los programas normalmente son un conjunto de algoritmos y datos externos. Siguiendo la programación estructurada:

```
struct punto_t {
    double x = 0.0;
    double y = 0.0;
};

class centroide {
    public:
    punto_t compute(const punto_t lp[], int nPuntos){
        punto_t pres;
        double medx = 0.0, medy = 0.0;
        /* computar el centroide*/
        return pres;
    }
    void print(punto_t p) {
        printf("(%f,%f)\n", p.x, p.y);
    }
};
```

Para este código, el Centroide no tiene estado. Tiene capacidades pero no tiene información interna. Entonces, si queremos añadir una dimensión de estados, se realizan modificaciones.

```
struct punto_t {
    double x = 0.0;
    double y = 0.0;

    void print(){
        printf("(%f,%f)\n",x,y); //además de un estado, ahora
           tiene la capacidad de pintarse.
    }
};
```

Entonces, ahora, donde si situarán los estados? Se define un gestor:

```
class gestor_puntos_2D{
    //define estados
    std:vector<punto_t>puntos; //puntos almacena un vector de
       punto_t
}
```

Tenemos entonces un gestor que gestiona el estado de un conjunto de puntos. ¿Qué capacidades le damos? Queremos calcular el Centroide con los datos del estado interno (antes los datos se proporcionaban de manera externa). Entonces, tenemos que modificar el prototipo:

```
punto_t centroide(){ //El centroide no necesita información
   porque ya la tiene disponible
    punto_t pres;
    double medx = 0.0, medy = 0.0;

    /* computar el centroide*/


    return pres;
}
```

El gestor devuelve la localización del Centroide.

```
#include <iostream>

int maquina_de_estados() {
    static auto estado = 0;
    return estado++;
}

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << maquina_de_estados() << std::endl;
    }
    return 0;
}
```

1. Sesión 1. Modificaciones menores de C a C++.

1.1. Mi primer programa en C++.

Vamos a ver un ejemplo inicial y una cuestiones fundamentales:

```
#include <iostream>                 //Cuestión 1
int main() {
    std::cout << "Hola mundo\n"; //Cuestión 2 y 3
}
```

1. Primera cuestión:

   ```
   #include <iostream>
   ```

   - `iostream` es una biblioteca (realmente, un archivo) que se incluye para operaciones de entrada y salida estándar. No lleva la extensión `".h"`, ya que forma parte de las bibliotecas modernas de C++.
   - Los caracteres `"<>"` se usan para buscar archivos ubicados en los directorios estándar del compilador, es decir, bibliotecas estándar.
   - Los caracteres `" "` se usan para incluir archivos que se encuentran en rutas relativas al proyecto.
   - C++ permite utilizar bibliotecas heredadas de C, pero con una convención moderna: se antepone una "c" al nombre del archivo y se omite la extensión `".h"`.

     ```
     #include <stdio.h> /*se convierte en*/ #include<cstdio>
     ```

2. Segunda cuestión:

   - Para escribir comentarios, se usan los caracteres `//`.
   - **Regla de oro: los comentarios deben estar alineados con el código al que hacen referencia**. Ejemplo:

     ```
     if(a == 3) { // la indentación es importante
         b = a + c;
         // aquí comento mi código
     }
     ```

   - Los comentarios **no se pueden anidar**.

3. Tercera cuestión:

   ```
   std::cout << "Hola mundo\n"; // Imprime un mensaje
   ```

   - Esta línea indica: "de la familia `std`, usamos el subgrupo `cout`". `cout` es el flujo de salida estándar, usado para imprimir texto en pantalla.
   - Se emplea el operador `"::"`, llamado operador de resolución de ámbito (`scope operator`), que permite acceder a elementos dentro de un espacio de nombres.

- El operador "<<" envía datos al flujo `cout` y se llama operador de inserción. También puede usarse para operaciones a nivel de bits. Ejemplo:

```
int a = 5 << 2; // Desplaza 2 bits el número 5: de 0101 a
   010100 (20 en decimal).
```

- El carácter especial "\n" indica un salto de línea. Como alternativa, se puede usar `std::endl`, que además de hacer un salto de línea, fuerza el vaciado del búfer de salida. Ejemplo:

```
#include <iostream>
int main() {
    std::cout << "Hola\n";
    std::cout << "Mundo" << std::endl;
    std::cout << "texto";
    return 0;
}
```

Ejemplo:

```
#include <iostream>
void main() {
    int a, b;  // Se declaran las variables 'a' y 'b'
    if (a == 3) {  // Condición: Si 'a' es igual a 3
        b = 5;  // Asigna 5 a 'b' si a=3
        std::cout << "hola";  // Muestra "hola"
    }
    else {  // Si la condición anterior no se cumple
        b = 6;  // Asigna 6 a 'b'
        std::cout << "Dos";  // Muestra el mensaje "Dos"
    }
}
```

Regla de oro: no repetir código en C++.

1.2. Ficheros en C++.

Extensiones de ficheros (varias opciones):

```
file.cpp
```

```
file.c++
```

```
file.cc
```

Extensiones de headers (varias opciones):

```
file.h
```

```
file.hpp
```

```
file.
```

1.3. Tipos de flujos en C++.

En C++, los flujos estándar se utilizan para la entrada y salida de datos. A continuación, se describen algunos de los flujos más comunes:

- `std::cin` es el flujo de entrada estándar en C++, asociado a `stdin`. Se utiliza para leer datos desde la entrada estándar, como el teclado. A los flujos de entrada les asociamos el operador "`>>`". Ejemplo sencillo:

```
#include <iostream>
void main() {
    int a;
    std::cin >> a; //Almacena cin en a
}
```

  Es concatenable, de izquierda a derecha. Ejemplo:

```
#include <iostream>
int main() {
    int i = 0;                          /*(1)*/
    double d = 0.0;                     /*(2)*/
    std::cin >> i >> d;                 /*(3)*/
    std::cout << "el valor de i es " << i << " y de d " <<
       d << std::endl;  /*(4)*/
    return 0;
}
```

  Comentarios a realizar:

  - Línea (1). Se declara una variable entera `i` e inicializa con el valor 0.
  - Línea (2). Se declara una variable de tipo `double` llamada `d` e inicializa con 0.0.
  - Línea (3). Se leen dos valores desde la entrada estándar: el primero se almacena en `i` y el segundo en `d`.
  - Línea (4). Se imprime el contenido de `i` y `d` utilizando la salida estándar con `cout`.

- `std::cout` es el flujo de salida estándar en C++, asociado a `stdout`. Se usa para enviar datos a la salida estándar, generalmente la pantalla (o impresora, I2C, etc.). A los flujos de entrada les asociamos el operador "`<<`" de inserción. Cabe mencionar, que un uso adicional de este operador es la sobrecarga de operadores, cuyo objetivo es cambiar el comportamiento de los operadores que deseemos (este uso se verá en un futuro). Ejemplo sencillo:

```
#include <iostream>
void main() {
    int a;
    std::cin >> a;
    std::cout << "El valor de a es: " << a << std::endl;
}
```

Es concatenable, de izquierda a derecha. Ejemplo:

```
#include <iostream>

void main() {
    int a = 5;
    int b = 10;
    // Concatenación de múltiples valores con cout
    std::cout << "El valor de a es: " << a << ", y el valor
       de b es: " << b << std::endl;
    // La salida: El valor de a es 5, y el valor de b es 10
}
```

- `std::clog` es el flujo de salida para los mensajes de log, asociado a guardar los datos de ejecución y a `stderr`. Este flujo está destinado a mensajes de error o advertencias, pero con un comportamiento diferente al de `std::cerr`. Cuando se usa `std::clog`, el mensaje se almacena en un búfer, lo que significa que la salida puede no ser inmediata.
- `std::cerr` es el flujo de error estándar en C++, también asociado a `stderr`. A diferencia de `std::clog`, cuando se imprime en `std::cerr`, los datos se envían inmediatamente al sistema operativo sin pasar por un búfer. Esto asegura que los mensajes de error se muestren al instante, sin ser retrasados por el mecanismo de almacenamiento temporal.

En conclusión, tanto `std::cin`, `std::cout` y `std::clog` son flujos buffered, que pueden tener una salida no instantánea. El `std::cerr` no lo es y fuerza una salida instantánea. Además,en los flujos `std::cin` y `std::cout`:

- Su comportamiento ya está definido para datos fundamentales.
- Son concatenables de izquierda a derecha.
- Modifican las variables directamente.

Ejemplo adicional:

```
#include <iostream>

int main() {
    int a = 8;
    const char cadena[] = "Hola";
    double b = 5.4;
    std::cout<<"texto: "<<cadena<<" a = " << a << " y ";
    std::cout<<"b = "<<b<<std::endl;
    return 0;
}
```

Salida por consola:

```
texto: Hola a = 8 y b = 5.4
```

1.4. El funcionamiento del búfer.

Un **búfer** es un área de memoria que almacena temporalmente los datos antes de enviarlos al sistema operativo. El propósito del búfer es optimizar el rendimiento al acumular datos y enviarlos en bloques, en lugar de uno a uno. Esto permite que el sistema opere de manera más eficiente, ya que se reduce el número de interacciones con el sistema operativo.

El **ritmo de envío de datos** se adapta gracias al uso de los búferes, lo que significa que no se afecta el proceso de escritura o lectura, ya que los datos se almacenan temporalmente y se vacían de manera progresiva. Sin embargo, cuando el búfer se llena demasiado, el sistema puede "bloquearse" y esperar a que el búfer se vacíe antes de continuar, lo que puede causar retrasos si el programa produce una gran cantidad de datos rápidamente.

Afortunadamente, la capacidad del búfer es generalmente conocida y controlada por el sistema operativo, lo que permite manejar adecuadamente el almacenamiento temporal sin que el sistema se vea sobrecargado.

1.5. El uso de `using`.

```
#include <iostream>
using namespace std;                    /*(1)*/

void main() {
    int entero;
    double real;
    char palabra[10] = "Hola";          /*(2)*/
    cout << "Un texto: " << palabra << endl;
    cin >> entero;
    cin >> real >> palabra;
    cout << "Real: " << real << "\nEntero: " << entero << "\n";
    cerr << "Palabra: " << palabra; /*(3)*/
}
```

Comentarios a realizar:

1. Línea (1). `using namespace std` importa todos los identificadores del espacio de nombres `std` y por tanto, nos ahorra escribir `std::cout` todo el rato. Si queremos importar solo unos símbolos determinados, se indican separados por comas sin orden concreto. Es tal que la línea anterior puede ser sustituida por:

   ```
   using std::cin, std::cout, std::endl, std::cerr;
   ```

   El funcionamiento sería el mismo, aunque es mucho más correcto. ¿Por qué? Importar todo un espacio de nombres arrastra todos los símbolos de ese espacio de nombres al espacio de nombres global. Esto puede dar lugar a conflictos de nombres y ambigüedades y, en algunos casos, incluso a errores que sólo se manifiestan en el tiempo de ejecución y son muy difíciles de detectar. En general, solamente recomendamos importar todo un espacio de nombres para

programas cortos y de uso temporal. Para programas largos y complejos, no se recomienda y tampoco en ficheros de cabecera que vayan a ser usados por múltiples unidades.

2. Línea (2). Esta línea es un ejemplo de una cadena de caracteres, `palabra` tiene 10 caracteres donde los vacíos se rellenan con ceros. Su contenido es:

```
{'H','o','l','a','\0','0','0','0','0'}
```

   Una alternativa sería escribir:

```
char palabra[] = "Hola"; k
```

   Se define una secuencia de caracteres de tamaño indefinido y la ventaja es que el propio compilador define el tamaño necesario para almacenar la palabra. Por tanto, en este caso su contenido sería:

```
{'H','o','l','a','\0'}
```

3. Línea (3). Salida inmediata debido a `cerr`.

Otro uso de `using` es para definir nuevos tipos de datos, equivalente a `typedef`:

```
using entero = int;
```

Esta línea de código equivale a decir: "Cada vez que te encuentres la palabra `entero`, sustitúyela por `int`". La forma equivalente de escribirlo usando `typedef` sería:

```
typedef int entero;
```

Ambas formas son funcionalmente equivalentes en este caso, pero `using` es más moderno y más flexible, especialmente cuando trabajamos con plantillas (`templates`).

Puntero a un entero:

```
using puntero = int *;
```

Equivalente para cada caso:

```
typedef int * puntero;
```

Array de 3 enteros:

```
using vector = int[3];
```

```
typedef int vector[3];
```

Vector de 3 punteros a enteros:

```
using vectorp = int * [3];
```

```
typedef int * vectop[3];
```

Puntero a grupos de 3 enteros:

```
using p_a_vector = int (*) [3];
```

```
typedef int (*p_a_vector)[3];
```

Recibe dos enteros y retorna `void`, `func` es un alias para un puntero a función:

```
using func = void (*)(int,int);
```

```
typedef void (*func)(int,int);
```

Figura 1: Otros ejemplos.

1.6. Anexo. Palabras clave en C++.

- auto
- bool
- break
- case
- catch
- char
- class
- const
- const_cast
- continue
- default
- delete
- do
- double
- dynamic_cast
- else
- enum
- explicit
- export
- extern
- false
- float
- for
- friend
- goto
- if
- inline
- int
- long
- mutable
- namespace
- new
- operator
- private
- protected
- public
- register
- reinterpret_cast
- return
- short
- signed
- sizeof
- static
- static_cast
- struct
- switch
- template
- this
- throw
- true
- try
- typedef
- typeid
- typename
- union
- unsigned
- using
- virtual
- void
- volatile
- wchar_t
- while

No son palabras clave:

- main
- cout
- cin
- endl
- string
- size_t
- NULL
- nullptr
- override
- final
- this
- std
- vector
- map
- mutex
- thread
- function
- constexpr
- typename
- decltype

2. Sesión 2. Variables en C++.

2.1. Concepto.

Una variable es un espacio de memoria al que se le ha asignado un tipo de elemento.

Para declarar una variable, lo que necesitamos es indicar el tipo básico de datos y el identificador.

El tipo de una variable nos afecta en el tamaño y las operaciones asociadas.

C++ es un lenguaje fuertemente tipado. Quiere decir que antes de usar una variable, tengo que indicar lo que va a contener.

Todo lo que comparte un tipo básico de datos se puede declarar en la misma línea y se crean las variables de izquierda a derecha. Ejemplo:

```
#include <iostream>
int main() {
    int x = 2, y = 2*x*x;
    std::cout<<x<<" e
       "<<y<<"\n";
    x = 4;
    std::cout<<x<<" e "<<y;
    return 0;
}
```

El ámbito de una variable es el bloque en el que está. El ámbito queda determinado por los caracteres "{ }" y define la zona en la que afecta la vida y la visibilidad de una variable.

- Variables locales.
  - Se crean cuando entran en el bloque.
  - Se inicializan cuando pasamos por la línea de declaración.
  - Se destruyen y liberan cuando salimos del bloque.
  - Su visibilidad es desde su declaración hasta el final del bloque.
- Variables globales.
  - Se crean al principio del programa.
  - Se inicializan al principio del programa.
  - Se destruyen al final del programa.
  - Su visibilidad es para las variables del mismo fichero y se ven desde su declaración hasta el final del fichero.
- Variables estáticas.
  - Se crean al principio del programa.
  - Se inicializan al principio del programa.
  - Se reinicializan la primera vez que se pasa por el código de declaración.
  - Se destruyen al final del programa.
  - Su visibilidad es desde su declaración hasta el final del bloque.

Ejemplo:

```
void func(3){
    static int i=0; //su valor perdura tras la primera
       compilación
    int j=0;
    cout<<i++<<j++;
}
void main() {
    func(); //resultado es 00
    func(); //resultado es 10
    func(); //resultado es 20
}
```

Reglas de oro:

- No puede haber dos identificadores iguales en el mismo ámbito.
- Siempre manda el identificador local frente a un identificador global. Es decir, **manda lo local sobre lo importado, y manda lo importado sobre lo global**.

2.2. Linkaje Externo e Interno.

En C y C++, los conceptos de **linkaje externo** e **interno** están relacionados con el alcance y la visibilidad de los símbolos (como variables y funciones) en un programa. A continuación, se explican ambos conceptos con ejemplos.

2.2.1. Linkaje Externo.

El **linkaje externo** permite que un símbolo global definido en un archivo fuente sea accesible desde otros archivos fuente dentro del programa. Características del linkaje externo:

- Los símbolos con linkaje externo tienen un **alcance global**.
- Las funciones y variables globales, por defecto, tienen linkaje externo.
- Para acceder a un símbolo externo en otro archivo, se utiliza la palabra clave `extern`.

Ejemplo de linkaje externo:

```
// Archivo1.cpp ------------------------------------------------
int valor = 42;  // Variable con linkaje externo por defecto
extern int valor = 42; //equivalente
// Archivo2.cpp ------------------------------------------------
extern int valor; // Declaración para usar la variable de
   Archivo1.cpp
void usarValor() {
    std::cout << valor << std::endl; // Acceso permitido
}
```

2.2.2. Linkaje Interno.

El **linkaje interno** restringe la visibilidad de un símbolo global al archivo donde se define, su funcionamiento es contrario a `extern`. Es útil para encapsular datos o funciones que no deben ser accesibles desde otros archivos. Características del linkaje interno:

- Los símbolos con linkaje interno son **privados** al archivo donde se definen.
- Se utiliza la palabra clave `static` para indicar que el símbolo tiene linkaje interno.
- Ayuda a evitar conflictos de nombres en proyectos grandes.

Ejemplo de linkaje interno:

```
// Archivo1.cpp ------------------------------------------------
static int contador = 0; // Linkaje interno, solo accesible en
   Archivo1.cpp
void incrementar() {
    contador++;
}
// Archivo2.cpp ------------------------------------------------
extern int contador; // Error: no se puede acceder al contador
```

Ejemplo 1 de clase:

```
// main.cpp ----------------------------------------------
#include <iostream>
using std::cout, std::endl;

// Declaración de funciones antes de su uso
void funcion1();
void funcion2();
void funcion11();
void funcion22();

// Variable global para el control de errores
int error_v = 0;

// Declaración y definición de variables globales extern
extern int var1 = 0;
extern int var2 = 0;

void errores() {
    cout << (error_v ? "Hay" : "No hay") << " errores" << endl;
}   /*(1)*/

int main() {
    extern int cuenta; /*(2)*/
    errores(); /*(3)*/
    for (int i = 0; i < 2; i++) {
        funcion1(); /*(4)*/
```

```
        funcion11(); /*(5)*/
        funcion2(); /*(6)*/
        funcion22(); /*(7)*/
    }
    cout << "Total de ejecuciones:" << cuenta << endl; /*(8)*/
    errores(); /*(9)*/
    return 0;
}
```

```
//funcion1.cpp -------------------------------------------
#include <iostream>

int cuenta;
extern int var1;
extern int var2 = 0;
void funcion1() {
    static int cuenta = 0; /*(10)*/
    std::cout << "funcion1:" << ++cuenta << std::endl; /*(11)*/
    ++::cuenta; /*(12)*/
}

void funcion11() {
    int cuenta = 0;
    std::cout << "funcion11:" << ++cuenta << std::endl;
    ::cuenta++;
    extern int error_v;
    error_v = 1;
}
```

```
//funcion2.cpp -------------------------------------------
#include <iostream>
using namespace std;

extern int cuenta;
void funcion22(); /*(13)*/

void funcion2() {
    cout << "funcion2" << endl; /*(14)*/
    cuenta++;
    funcion22();
}
static void funcion22() {
    cout << "funcion22 no cuenta\n"; /*(15)*/
}
```

Comentarios a realizar:

- Línea (1). Muestra "Hay errores" o "No hay errores" según el valor de `error_v`.
- Línea (2). Se declara como externa la variable global `cuenta` para registrar ejecuciones.

- Línea (3). Se llama a `errores` antes del bucle.
- Línea (4–7). Se llaman a las funciones definidas en otros archivos.
- Línea (8). Se imprime el total de ejecuciones acumuladas en `cuenta`.
- Línea (9). Se muestra el estado de errores tras la ejecución.
- Línea (10). Variable local estática: persiste entre llamadas a la función.
- Línea (11). Se imprime y se incrementa la variable estática `cuenta`.
- Línea (12). Incrementa la variable global `cuenta`.
- Línea (13). Declaración anticipada de la función `funcion22`.
- Línea (14). Se imprime un mensaje indicando la ejecución de `funcion2`.
- Línea (15). Se imprime un mensaje desde una función `static`, que sólo es visible dentro de este archivo.

Salida esperada:

```
No hay errores
funcion1: 1
funcion11: 1
funcion2
funcion22 no cuenta
funcion1: 2
funcion11: 1
funcion2
funcion22 no cuenta
Total de ejecuciones: 2
Hay errores
```

Ejemplo 2 de clase:

```
#include <iostream>
using std::cout, std::endl;
int val = 5;
int main() {
    int val = 3;
    while(::val<10){
        ++::val;
        for(int val = 0; val<5; val++)
        cout<<::val<<";"<<val<<endl;
        cout<<val<<endl;
    }
    return 0;
}
```

Comentarios a realizar:

- El operador scope :: es el más prioritario de C++ y se utiliza para hacer referencia a una variable global. De modo que:

```
++::val; //"incrementa en 1 el valor de la variable global
   val".
```

- El bucle flor solamente contiene una sentencia, luego no son necesarios los corchetes{ }.

```
for(int val = 0; val<5; val++)
cout<<::val<<";"<<val<<endl;
```

- Salida por consola:

```
6;0 -> 6;1 -> 6;2 -> 6;3 -> 6;4 -> 3 -> 7;0 -> etc...
```

2.3. Inicialización.

Inicializar es el valor que les damos a las variables cuando las creamos en la línea de declaración. Las variables fundamentales, se dividen en dos grupos:

- Variables automáticas. De estas se dice que no se inicializan.
- Variables estáticas o globales. Sí se inicializan y siempre utilizan `zero initializer`, es decir, tienen el valor 0 por defecto.

Es decir, normas:

- Todo lo creado tiene inicialización.
- No inicializar implica una inicialización por defecto.

```
int a;
new int;
new Cosa;
int v[10]; //todos los elementos son ceros
```

En C++11 existen 3 modos de inicializar:

- Inicialización de copia. Sintaxis:

```
tipo identificador = expresión
```

  Ejemplo:

```
int x=0;
```

- Inicialización directa. Sintaxis:

```
tipo identificador (expresión)
```

  Ejemplo:

```
int x(0); //Este modo es propio de C++03.
```

- Inicialización de listas o uniforme. Sintaxis:

```
tipo identificador {expresión}//Este modo es el más
   recomendado
```

  Ejemplos:

```
int x{3};
int v[10]{}; //vector vacío de 10 enteros y todos ceros
int v[10]{0,1,2} //Los elementos indefinidos son ceros.
int a{5};//int a = 5;
int a{}; //int a = 0;
```

Ejemplo:

```
#include <iostream>

using namespace std;

struct Punto {
    int x, y;
    void print(){cout<<"("<<x<<","<<y<<")";}
}pglobal;

void print(Punto p) {
    cout << "(" << p.x << "," << p.y << ")";
}

int main() {
    Punto plocal; // Basura (no inicializado)
    Punto plocal2{}; // Inicialización a 0
    static Punto plocal3; // Inicializado a 0

    print(pglobal);
    print(plocal);
    print(plocal2);
    print(plocal3);

    Punto otro{2, 3}; // (2,3)

    print(otro); // (2,3)
    otro.print();

    return 0;
}
```

Salida por consola:

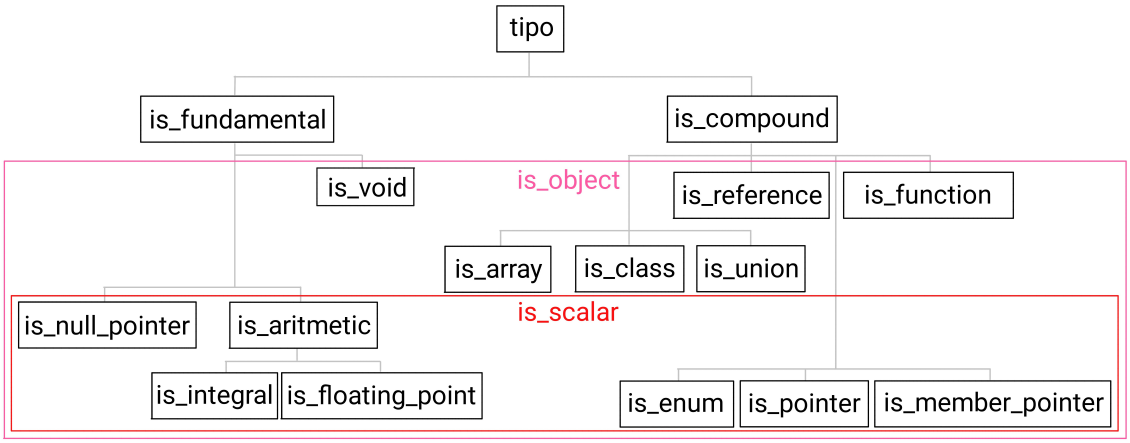
```
(0,0)(945596784,32764)(0,0)(0,0)(2,3)(2,3)
```

3. Sesión 3. Tipos de datos en C++.

3.1. Concepto.

Partimos de los datos contenidos en la librería:

```
#include <type_traits>
```



El diagrama muestra cómo los diferentes tipos en C++ pueden clasificarse jerárquicamente usando los traits disponibles en `<type_traits>`. A continuación se detallan estas clasificaciones:

- `is_fundamental`: Determina si un tipo es un tipo fundamental (como `int`, `float`, `char`, `void`, etc.).
  - `is_void`: Identifica el tipo `void`.
  - `is_arithmetic`: Identifica los tipos numéricos.
    - `is_integral`: Identifica tipos enteros (`int`, `char`, etc.).
    - `is_floating_point`: Identifica tipos de punto flotante (`float`, `double`, etc.).
- `is_compound`: Clasifica los tipos más complejos o compuestos.
  - `is_reference`: Identifica referencias (`T&`, `T&&`).
  - `is_function`: Identifica funciones.
  - `is_object`: Identifica cualquier tipo que no sea ni función, ni referencia, ni `void`.
- `is_object`:
  - Agrupa tipos como:
    - `is_array`: Identifica arreglos.
    - `is_class`: Identifica clases.
    - `is_union`: Identifica uniones.

- `is_scalar`:
  - Incluye tipos que pueden representarse como valores simples:
    - `is_null_pointer`: Identifica el puntero nulo (`nullptr`).
    - `is_enum`: Identifica enumeraciones.
    - `is_pointer`: Identifica punteros.
    - `is_member_pointer`: Identifica punteros a miembros.

La utilidad de esta librería es que nos permite usar funciones para verificar tipos, modificar tipos y hacer programación condicional en base a los tipos de datos. En un principio, en esta asignatura no se indaga mucho en este tema, así que, este es un ejemplo sencillo para dar contexto a este apartado:

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

int main() {
    cout << boolalpha; /*(1)*/
    cout << "int es un entero?: "<< is_integral<int>::value
       <<"\n"; /*(2)*/
    cout << "double es un entero?: "<<
       is_integral<double>::value <<"\n"; /*(3)*/
}
```

Comentarios a realizar:

- Línea (1). `boolalpha` en la salida de `cout` convierte los valores booleanos true y false para que se impriman como texto "true" o "false" en lugar de 1 o 0.
- Línea (2). Ya que `int` es un tipo integral y `is_integral<int>::value` devuelve `true`.
- Línea (3). Ya que `double` no es un tipo `integral` y `is_integral<double>::value` devuelve `false`.

3.2. Función Static Assert.

`Static Assert` es una función que "si lo que hay dentro no se cumple, no compiles". Una situación posible es que si un tipo de datos no es entero, entonces no compila. Se puede usar tanto fuera como dentro del main. Una manera de comprobar si la máquina es de 64 bits o no:

```
static_assert(sizeof(int*)) == 8 // Este código es para 64 bits.
```

3.3. Tipo de datos `void`.

Ideas clave:

- El tipo de datos `void` es un tipo de datos incompleto.
- No se puede crear una variable con este tipo de dato.

```
void a; //error
```

- No se puede crear una referencia con este tipo de dato.

```
void&b(a); //error
```

- No se pueden crear vectores con `void`.

```
void v[10]; //error
```

- La primera utilidad que tiene es crear funciones que no retornan nada, funciones que no toman datos y que pueden devolver `void*`.
- La segunda utilidad que tiene es crear punteros que apuntan a nada. Estos punteros se conocen como punteros vacíos o punteros puros, los cuales usaremos en el futuro.

```
int a;
void *p = &a; /*(1)*/
```

Línea (1). Quiere decir que es un puntero que tiene la dirección de la variable `a` y que no apunta a nada en específico.

3.4. Tipo de datos `nullptr`.

Tipo de datos especial que permite ser asignado a cualquier tipo de datos. Es un 0 pero no se puede operar con él. Lo usamos para definir un puntero que no apunta a nada. Ejemplos equivalentes:

```
char *a = nullptr;
```

```
char *a{};
```

Los punteros cuando se inicializan, usan `nullptr` para compararlo o hacer inicialización nula. Ejemplos de código equivalente:

```
#include <iostream>
int main(){
    int a;
    decltype(a)b;
    return 0;
}
```

```
#include <iostream>
int main(){
    int a;
    int b;
    return 0;
}
```

3.5. Operadores de tipos de datos fundamentales.

- Aritméticos.

Operador	Descripción	Ejemplo (a=5, b=2)	Resultado
`+`	Suma	`a + b`	7
`-`	Resta	`a - b`	3
`*`	Multiplicación	`a * b`	10
`/`	División entera	`a / b`	2
`%`	Módulo (Residuo)	`a% b`	1

Además se pueden combinar y reescribir como:

Operador	Descripción	Ejemplo (a=5, b=2)	Resultado de a
`+=`	Suma y asignación	`a += b`	7
`-=`	Resta y asignación	`a -= b`	3
`*=`	Multiplicación y asignación	`a *= b`	10
`/=`	División y asignación	`a /= b`	2
`%=`	Módulo y asignación	`a%= b`	1

- Unitarios:

Operador	Descripción	Ejemplo (a=5)	Resultado de a
`++a`	Preincremento	`b = ++a`	`a=6, b=6`
`a++`	Postincremento	`b = a++`	`a=6, b=5`
`--a`	Predecremento	`b = --a`	`a=4, b=4`
`a--`	Postdecremento	`b = a--`	`a=4, b=5`

  - `+a` : se utiliza para convertir a enteros. Ejemplo:

```
#include <iostream>
int main() {
    char a = 'A';
    std::cout << +a;
    return 0;
}
```

- `-a` : se utiliza para invertir el signo. Ejemplo:

```cpp
#include <iostream>

int main() {
    int a = 5;
    std::cout << -a; // Cambia el signo de 'a' y muestra -5
    return 0;
}
```

- Comparaciones.

Operador	Descripción	Ejemplo (a=5, b=10)	Resultado
`==`	Igual a	`a == b`	`false (0)`
`!=`	Diferente de	`a != b`	`true (1)`
`<`	Menor que	`a <b`	`true (1)`
`>`	Mayor que	`a >b`	`false (0)`
`<=`	Menor o igual que	`a <= b`	`true (1)`
`>=`	Mayor o igual que	`a >= b`	`false (0)`

Ejemplo en C++:

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

int main() {
    int a = 5, b = 10;
    cout << (a == b) << endl;  // false (0)
    cout << (a != b) << endl;  // true (1)
    cout << (a < b) << endl;   // true (1)
    cout << (a > b) << endl;   // false (0)
    cout << (a <= b) << endl;  // true (1)
    cout << (a >= b) << endl;  // false (0)
    return 0;
}
```

- Lógica booleana:

Operador	Descripción	Ejemplo (a=true, b=false)	Resultado
`&&`	AND lógico (conjunción)	`a && b`	`false` (0)
`||`	OR lógico (disyunción)	`a || b`	`true` (1)
`!`	NOT lógico (negación)	`!a`	`false` (0)

Ejemplo en C++:

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

int main() {
    int a = 10, b = 5;
    cout << "AND lógico (a > 5 && b < 10): " << (a > 5 && b <
       10) << endl;
    cout << "OR lógico (a == 5 || b == 5): " << (a == 5 || b ==
       5) << endl;
    cout << "NOT lógico (!(a == 10)): " << !(a == 10) << endl;
    return 0;
}
```

- Bit a bit (bitwise).

Operador	Nombre	Descripción
&	AND bitwise	Devuelve 1 si ambos bits son 1
|	OR bitwise	Devuelve 1 si al menos un bit es 1
^	XOR bitwise	Devuelve 1 si los bits son diferentes
~	NOT bitwise	Invierte todos los bits
&=	AND compuesto	Realiza una operación AND y asigna el resultado
|=	OR compuesto	Realiza una operación OR y asigna el resultado
^=	XOR compuesto	Realiza una operación XOR y asigna el resultado

Ejemplo:

```
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 3;
    cout << "a & b: " << (a & b) << endl;  // AND
    cout << "a | b: " << (a | b) << endl;  // OR
    cout << "a ^ b: " << (a ^ b) << endl;  // XOR
    cout << "~a: " << (~a) << endl;        // NOT
    return 0;
}
```

- Operadores desplazamiento.

Operador	Nombre	Descripción
<<	Desplazamiento izq.	Desplaza bits a la izquierda
>>	Desplazamiento der.	Desplaza bits a la derecha
<<=	Desplazamiento izq.	Desplaza bits a la izquierda con asignación
>>=	Desplazamiento der.	Desplaza bits a la derecha con asignación

Ejemplo 1:

```
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 3;
    cout << "a << 1: " << (a << 1) << endl; // Desplazamiento
       izq
    cout << "a >> 1: " << (a >> 1) << endl; // Desplazamiento
       der
    return 0;
}
```

Ejemplo 2:

```
#include <iostream>
using namespace std;
int main() {
    int a;
    if((cin>>a)&&(a<4)){
        cout << "hola";
    }
    return 0;
}
```

3.6. Relación entre tipos de datos

Pregunta: ¿Qué ocurre si mezclamos tipos de datos?

- `int + real`: se promociona el entero a real. Ejemplo:

```
#include <iostream>

int main() {
    std::cout<<2/4<<std::endl;
    std::cout<<2/4.0;
    return 0;
}
```

- `dos real o dos integral`: en los dos reales, se pasa al más grande, con los integrales pasa lo mismo
- Para las operaciones aritméticas, los integrales que son menores que `int`, pasan a ser `int`. Ejemplo:

```
int a = false + true; //suma de dos int
```

- `unsigned` con `signed` mucho cuidado con estas comparaciones.

Ejemplo:

```
#include <iostream>
#include <cstdint>
using std::cout;
using std::endl;

int main() {
    int32_t num = 0x45F1D56;
    cout << num << "--> "<<std::hex<<num<<endl;
    uint8_t a = num&0xFF,
    b = (num&0xFF00)>>8,
    c = (num&0xFF0000)>>16,
    d = (num>>24);
    cout << +d << " " << +c << " "<< +b << " " << +a << endl;
    if(d&0x80) cout << "ES NEGATIVO \n"; //comprueba el bit d
       tenga el numero 8
    int32_t litend = a|b << 8|c << 16|d<<24, bigend = d|c <<
       8|b << 16|a <<24;
    cout << litend << "--"<< bigend << endl;
    cout << std::dec << litend << "--" << bigend << endl;
}
```

Cuestión de clase: De estos siguientes programas, ¿Cuáles compilan?

```
//Programa A
#include <iostream>
void main() {
    float sqrt;
    sqrt = 8.0F;
    std::cout<<sqrt;
}

//Programa B
#include <iostream>
void main() {
    float auto; //auto es palabra clave
    auto = 8.0F;
    std::cout<<auto;
}

//Programa C
#include <iostream>
using std::cin,std::cout;
void main() {
    float cin; //manda lo local
    cin = 3.0;
    cout<<cin;
}
```

Respuesta:

- A compila.
- B no compila.
- C compila.

3.7. Métodos de la librería estándar.

La librería estándar de C++ (`std`) proporciona un conjunto extenso de clases, funciones y objetos esenciales para el desarrollo de programas. A continuación, se presentan sus principales funcionalidades:

• Entrada y Salida (I/O): Ofrece herramientas para la comunicación con el usuario.

- `std::cout` — Salida estándar.
- `std::cin` — Entrada estándar.
- `std::cerr` — Salida de errores.
- `std::clog` — Mensajes de registro.

• Estructuras de Datos y Contenedores: Facilita el almacenamiento y manipulación de datos.

- `std::vector`, `std::list`, `std::deque` — Contenedores de datos.
- `std::stack`, `std::queue` — Estructuras LIFO y FIFO.
- `std::set`, `std::map` — Contenedores de elementos *clave-valor*.

• Algoritmos: Proporciona funciones para ordenar, buscar y manipular datos.

- `std::sort` — Ordenación de elementos.
- `std::find` — Búsqueda en un rango.
- `std::min` y `std::max` — Mínimo y máximo.

• Funciones Matemáticas: Realiza operaciones y cálculos numéricos.

- `std::pow` — Potenciación.
- `std::sqrt` — Raíz cuadrada.
- `std::abs` — Valor absoluto.

• Manejo de Cadenas (Strings): Facilita la manipulación de texto.

- `std::string` — Manejo de cadenas de texto.
- `std::to_string` — Conversión de números a texto.
- `std::stoi` y `std::stod` — Conversión de texto a números.

• Manipulación de Iteradores: Permite recorrer estructuras de datos.

- `begin()` y `end()` — Inicio y fin del recorrido.
- `rbegin()` y `rend()` — Recorrido inverso.

• Multihilos (Threading): Facilita la ejecución paralela de tareas.

- `std::thread` — Creación de hilos.
- `std::mutex` — Sincronización de recursos.

3.8. Ejercicio de clase. Función Raíz cuadrada.

Enunciado del Ejercicio

Desarrolla un programa en C++ que calcule la raíz cuadrada de un número real positivo utilizando el Método de bisección. Este método se basa en acotar progresivamente el intervalo donde se encuentra la solución, analizando el signo del punto medio en cada iteración.

Requisitos del Ejercicio

1. Inicialización del Intervalo:
   - Si el número de entrada $x > 1$, el intervalo inicial debe ser $[1, x]$.
   - Si $0 < x < 1$, el intervalo inicial debe ser $[x, 1]$.
2. Algoritmo de Bisección:
   - Calcula el punto medio del intervalo actual:
   $$m = \frac{\text{inferior} + \text{superior}}{2}$$
   - Actualiza los límites del intervalo según el resultado:
     - Si $m \times m > x$, entonces superior $= m$.
     - Si $m \times m < x$, entonces inferior $= m$.
3. Criterio de Finalización:
   - El proceso se repite hasta que la diferencia entre superior e inferior sea menor que una precisión definida por el usuario (por ejemplo, 1×10^{-6}).

Salida Esperada

El programa debe mostrar en pantalla el valor aproximado de la raíz cuadrada del número ingresado, respetando la precisión especificada.

Solución:

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

double raiz_cuadrada(double x);
double puntomedio(double a, double b);

int main() {
    cout << "Dame un numero: ";
    double x;
    cin >> x;
    cout << "La raiz es " << raiz_cuadrada(x);
    return 0;
}

double raiz_cuadrada(double x) {
    double superior{}, inferior{}, m{};
    superior = x + 1; //la suma de los intervalos es [0,x+1]
    m = puntomedio(superior, inferior);
    while (fabs(superior - inferior) > 1e-6) {
        if (m * m > x)
        superior = m;
        else
        inferior = m;
        m = puntomedio(superior, inferior); // IMPORTANTE
    }
    return superior;
}

double puntomedio(double a, double b) {
    double middle{};
    middle = (a + b) / 2.0;
    return middle;
}
```

4. Sesión 4. La clase `vector`.

Un objeto `vector` es un contenedor que pertenece a la librería STL (librería estándar de plantillas).

```
#include <vector>
using std::vector;
```

En la realidad, un `vector` es una metaclase porque se define una lógica general de "vector" que se aplica a la clase `vector` que corresponda. Un `vector` se puede especificar para tipos de datos `int,char,vector,...`

Características de los vectores:

- Un `vector` almacena objetos del mismo tipo.
- En la memoria, se almacenan consecutivamente los objetos (sin espacios entre estos mismos).
- Los objetos se acceden por indexación `[i]` entre `0` y `N-1`, siendo `N` la dimensión del `vector`.
- Pueden existir vectores vacíos.
- Puede crecer/reducir el tamaño de un `vector`.

4.1. Inicialización de la clase `vector`.

- Inicialización por defecto o vacía. Ejemplos equivalentes:

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

- Inicialización con una serie de valores diferentes. Ejemplos equivalentes:

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

- Inicialización de un vector de dimensión n con sus elementos inicializados por defecto o a cero en el caso de tipos fundamentales.

```
vector<int> v(4); //vector con 4 enteros nulos
```

- Inicialización de un vector de dimensión n con todos sus elementos igual a un valor indicado.

```
vector<int> v(4,3); //vector con 4 enteros inicializados a 3
```

- Inicialización de un vector como copia de otro `vector` (un duplicado elemento a elemento).

```
vector<int> v5 = {1, 2, 3, 4, 5};
vector<int> v6(v5);  // v6 es una copia de v5
vector<int> v7 = v5; // Equivalente a lo anterior
```

4.2. Métodos de la clase `vector`.

- `v.size()` devuelve `size_t` que es un tipo de datos de naturaleza `unsigned int`. Esta función determina cuántos elementos tiene actualmente un vector.
- `v.capacity()` devuelve el número de elementos que cabrían en el espacio de memoria reservado. Al iniciar el vector, el tamaño `size` y la capacidad `capacity` son los mismos. Al borrar un elemento, el tamaño disminuye pero la capacidad permanece igual. Es decir, si el vector tiene inicialmente "n" elementos y eliminas uno, el `size` será "n-1" pero la `capacity` seguirá siendo "n".
- `v.front()` devuelve una referencia al primer miembro del vector. Esto permite tanto leer como modificar el valor.
- `v.back()` devuelve una referencia al último miembro del vector. Esto permite tanto leer como modificar el valor.
- `v[i]` con `i` entre `[0, n]`. Accede al elemento en la posición `i`.
- `v.at(i)` similar a `v[i]`, pero con comprobación de límites. Si el índice está fuera del rango válido, lanza una excepción del tipo `std::out_of_range`.
- `v.push_back(objeto)` introduce un objeto (del mismo tipo que los elementos del vector) al final de este, incrementando el tamaño en 1.
- `v.pop_back()` elimina el último elemento, reduciendo el tamaño en 1 pero no cambia la capacidad.
- `v.clear()` elimina todas los elementos del vector, dejando el tamaño en 0, pero no modifica la capacidad.
- `v.empty()` devuelve `true` si el vector está vacío, de lo contrario, devuelve `false`.

Ejemplo 1:

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

int main() {
    std::vector<int> v{2, 4, 7, 5, 3, 1, 4, 3};
    std::cout << v.size() << " " << v.capacity() << "\n";
    v[3] = 9;

    v.push_back(15);            /*(1)*/
    std::cout << v.size() << " " << v.capacity() << "\n";

    for (int i = 0; i < v.size(); i++) { /*(2)*/
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;

    for (int y: v) {
        std::cout << y << " "; /*(3)*/
    }
    std::cout << std::endl;

    for (auto y: v) {           /*(4)*/
        std::cout << y << " ";
    }
    std::cout << std::endl;    /*(5)*/

    return 0;
}
```

Salida por consola:

```
8 8
9 16
2 4 7 9 3 1 4 3 15
2 4 7 9 3 1 4 3 15
2 4 7 9 3 1 4 3 15
```

Comentarios a realizar:

- Línea (1). Añade un 15 después del final del vector aumentándolo de tamaño.
- Líneas (2) y (3). Forma no recomendada.
- Línea (4). Equivalente a la forma anterior pero utilizando ventajas que nos da C++. Esto nos deja la ventaja de no tener que saber el tipo de vector que es.
- Línea (5). Hay algunas veces que necesitamos saber el índice del valor al que estamos accediendo y de esta manera no podemos saberlo.

Ejemplo 2:

```
#include <vector>
#include <iostream>
using std::cout;
using std::vector;
using std::endl;
using std::cin;

int main() {
    vector <int> v{2,7,9};       /*(1)*/
    cout << v.size() << endl;    /*(2)*/
    cout << v[0] << " " << v[1] << endl;
    v[0] = 5;
    v[1]++;
    v[2] *= 2;
    int *p = &v[1];
    cout << v.front();           /*(3)*/
    cout << v.back();            /*(4)*/
    v.back() = 8;                /*(5)*/
    p = &v.back();               /*(6)*/
    return 0;
}
```

Salida por consola:

```
3
2 7
518
```

Comentarios a realizar:

- Línea (1). Se crea una variable `v` de tipo vector de 3 enteros inicializada.
- Línea (2). Se pregunta por el tamaño del vector mediante la función `v.size()`. El tamaño es 3.
- Línea (3). Imprime el primer elemento del vector.
- Línea (4). `v.back()` devuelve una referencia al valor del último miembro del `vector` e imprime su valor, una alternativa sería escribir:

  ```
  v[v.size() - 1]
  ```

  Una alternativa adicional sería escribir:

  ```
  if(v.size())
      cout << v[v.size() - 1];
  ```

- Línea (5). Cambia el valor del último elemento del vector a 8.
- Línea (6). Actualiza el puntero `p` para que apunte al último elemento.

Ejemplo 3:

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

int main() {
    std::vector<int> v = {2, 7, 9};      /*(1)*/

    std::cout << v.size() << std::endl;
    std::cout << v[2] << "," << v[1] << std::endl;

    v[0] = 3;
    v[1]++;
    v[2] += 3;

    int *p = &v[2];                      /*(2)*/

    std::cout << v.front() << std::endl;
    std::cout << v.back() << std::endl;

    v.front() = 10;                      /*(3)*/
    return 0;
}
```

Salida por consola:

```
3
9,7
3
12
```

Comentarios a realizar:

- Línea (1). Se declara e inicializa un vector de enteros con los valores `{2, 7, 9}`.
- Línea (2). No hay acceso fuera de los límites.
- Línea (3). Se modifica el primer elemento usando `v.front() = 10`.

4.3. Recorrido de vectores.

4.3.1. Mediante for de rango C++11.

Se sigue la estructura:

```
for(declaración_de_rango:expresión_de_rango){
    sentencia;
}
```

- `declaración de rango` : es una declaración de una variable del mismo tipo básico del vector y que se utiliza para iterar a lo largo del vector.
- `expresión de rango` : es cualquier expresión o colección de estas que represente algo secuencial iterable.
- `sentencia` : expresión o colección de estas que se realizan durante el bucle.

Ejemplo:

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

int main() {
    //compilador sabe que hay 7 elementos en el array de C
    int x[] = {1,2,3,4,5,6,7};

    //Recorrido usando range-based for loop
    for (auto y:x) cout<<y<<' ';
    cout<<endl;

    //Recorriendo usando for tradicional
    for(int i=0; i<7; i++) cout<<x[i]<<' ';
    cout<<endl;

    //Recorriendo vector con range-based for loop
    vector<int> v{0,1,2,3,4};
    for(auto n:v) cout<<n<<' ';
    cout<<endl;

    //Recorriendo una lista inicializada directamente en el for
    for(auto n:{0,1,2,3,5,8,13,21}) cout<<n<<' ';
    return 0;
}
```

Adicionalmente, existen 3 formas de acceso a los elementos del vector según la forma en que se pasa el valor

1. Modo de acceso por copia. No se modifican los valores originales y en cada iteración se hace una copia del elemento iterado que se contiene en la variable declarada. Estructura:

```
for (auto variable:expresión_de_rango){
}
```

Ejemplo:

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

int main() {
    vector<int> nums = {1, 2, 3, 4, 5};

    for (auto n:nums) {
        cout << n << " ";
    }
    return 0;
}
```

2. Modo de acceso por referencia. Similar al modo anterior sin embargo, gracias a que usamos &, podemos acceder a los elementos del vector en vez de a una copia. Estructura:

```
for (auto &variable:expresión_de_rango){
}
```

Ejemplo:

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

int main() {
    vector<int> x{0,1,2,3,4};
    for(auto &y:x) y = 0;
    for(auto n:x) cout<<n<<' ';
    cout<<endl;
    return 0;
}
```

3. Modo de acceso por referencia constante. Este modo emplea una referencia constante y es útil cuando el vector tiene un tamaño grande con el fin de no gastar tanta memoria, ahorrando hacer un duplicado en cada ciclo. No permite modificar el valor de los elementos. Estructura:

```
for (const auto &variable:expresión_de_rango){
}
```

4.3.2. Mediante iteradores C++.

Se sigue la estructura:

```
for (auto i = vector.begin(); i != vector.end(); ++i) {
}
```

La idea es generalizar el indicador de ubicación de un elemento contenido, permitiéndonos acceder a este y el resto de elementos de un vector. Esta idea nos permite además generalizar algoritmos dado que el acceso a los elementos puede ser abstraído y aplicado a cualquier vector.

Las principales ventajas de este método es que permite estandarizar los procesos y la seguridad del comportamiento de los iteradores entre operaciones que modifiquen la estructura del propio vector.

En pocas palabras, un iterador es un tipo de datos definido por el propio vector que se usa para referir una ubicación dentro del mismo. Esto implica el uso de funciones como `begin()` y `end()`.

```
vector <int> v{1,2,3,4};
auto i = begin(v);
auto e = end(v);
```

Comentarios a realizar:

- La línea:

  ```
  auto i = begin(v);
  ```

  Es equivalente a :

  ```
  v.begin();
  ```

- Antes de C++11 , se tenía que escribir lo siguiente:

  ```
  std::vector <int>::iterator i = begin(v);
  ```

  Luego `auto` nos ha facilitado bastante la vida.

- La línea:

  ```
  auto e = end(v);
  ```

  Es equivalente a :

  ```
  auto e = v.end();
  ```

Los operadores que más usamos son los siguientes:

- `*` : acceder al contenido de la variable apuntada.
- `+` : incrementa la posición del iterador, apuntando a un nuevo elemento. Se recomienda revisar el apartado 3.5. En general, todos estos operadores sirven para cambiar la posición del iterador.

Ejemplo 1:

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

int main() {
    vector <int> v{1,2,3,4};
    auto i = begin(v);
    auto e = end(v);

    cout << *i<<endl;
    cout << *(i+2)<<endl;
    cout << *e<<endl; //Comportamiento indefinido

    for(auto i = begin(v); i!=end(v); ++i)
        cout<<*i<<' ';
    cout <<endl;
    for(size_t i = 0; i<v.size();  i++)
        cout<<v[i]<<' ';
    cout <<endl;
    for(size_t i = 0; i<v.size();  i++)
        cout<<v.at(i)<<' ';
    return 0;
}
```

Comentarios a realizar:

- Accede a el principio del vector.

  ```
  cout << *i<<endl; // v[0]
  ```

- Accede el elemento `v[2]`.

  ```
  cout << *(i+2)<<endl;
  ```

- `end(v)` devuelve un iterador apuntando después del último elemento, es decir, es fuera de los límites válidos. Luego, estás intentando acceder a memoria fuera del vector, lo que causa comportamiento indefinido.

  ```
  cout << *e<<endl;
  ```

Ejemplo 2:

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

int main() {
    vector <int> v{1,2,3,4};
    auto ini = begin (v);
    auto e = end(v);

    int v2[] = {1,2,3,4};
    auto ini2 = begin(v2);
    auto e2 = end(v2);

    cout <<*ini<<' '; //1
    cout <<*ini++<<' ';//imprime 1 aunque luego ini apunta a 2
    cout <<*ini<<' '; //2
    cout <<*++ini<<endl; //3
    --ini;
    cout<<*ini<<' '; //2
    ini += 2; //avanzar dos posiciones
    cout << *ini<<' '; //4
    ini -=3;
    cout<<*ini<<endl; // 1
    cout<<(e-ini)<<(e>ini)<<(ini==e)<<endl;

    //Recorrido con un for
    for (auto i = begin(v); i != end(v); ++i)
        cout <<*i<<' ';
    return 0;
}
```

Comentarios a realizar:

- Si resto `e-ini` entonces estoy buscando saber cuántos elementos hay entre `e` y `ini`. Es decir, buscar cuántos elementos quedan para llegar al final.

```
cout<<(e-ini)<<(e>ini)<<(ini==e)<<endl;
```

- La operación `e>ini` es para saber si está `e` delante de ini.

```
cout<<(e-ini)<<(e>ini)<<(ini==e)<<endl;
```

- **Regla de oro: nunca modificar el contenedor iterando con un iterador, porque normalmente el compilador optimiza las operaciones, luego no pregunta por el `end(v)` constantemente. Siempre modificar el vector fuera del iterador**.

```
for (auto i = begin(v); i != end(v); ++i)
    cout <<*i<<' ';
```

4.3.3. Mediante índices.

Se sigue la estructura:

```
for (size_t i = 0; i < v.size(); i++) {
}
```

Se caracteriza por ser claro y simple. Te da el índice, útil si necesitas saber la posición. Puedes acceder/modificar por posición.

Ejemplo:

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

int main() {
    vector <int> v{1,2,3,4};
    for(size_t i =0; i<v.size(); i++)
        cout<<v[i];
    return 0;
}
```

4.4. Datos regulares.

En C++, datos regulares son tipos que se comportan como los tipos nativos (como int, double, etc.). El término viene de la programación genérica (por ejemplo, en la STL), donde es deseable que los objetos se comporten de forma predecible y segura con operaciones básicas.

Un tipo regular de C++ debe tener:

- Constructor por defecto.
- Constructor de copia.
- Operador de asignación `operator=`
- Destructor.
- Operadores de comparación.

Cuando tú defines tus propias clases y quieres que sean regulares, necesitas implementar comportamientos que:

- Copien profundamente (`deep copying`).
- Asignen profundamente (`deep assignment`)
- Comparen profundamente (`deep comparison`)
- Posean propiedad completa de sus recursos (`deep ownership`)

Cuatro propiedades de los datos regulares:

- Deep copying: copia profunda, el elemento no preexiste.

```
auto v2 = v;
```

- Deep assignment: el elemento sí preexiste.

```
auto v3{};
v3 = v2;
```

- Deep comparison: todo lo de dentro es comparado. Se usa el operador `<=>` , no lo usamos realmente. Entendemos:

```
v2<v3; //irá comparando elemento a elemento el v2 con el v3
```

- Deep ownership.

4.5. Ejercicio de clase. Números Romanos

Enunciado del Ejercicio

Desarrolla un programa en C++ que convierta un número entero positivo a su representación en números romanos. El programa debe utilizar una función auxiliar que facilite la impresión de los símbolos romanos correspondientes a cada dígito según su posición decimal.

Requisitos del Ejercicio

1. **Funciones Principales:**
   - Implementa una función llamada `digito_facil` que reciba cuatro parámetros:
     - Un número entero d que representa el dígito a convertir.
     - Tres caracteres h, m, l que representan los símbolos romanos de "alto", "medio" y "bajo" respectivamente.
2. **Función de Conversión:**
   - Implementa una función llamada `imprime_romanos` que reciba un número entero positivo y lo convierta a números romanos utilizando la función `digito_facil`.
   - Esta función debe dividir el número en sus dígitos correspondientes (miles, centenas, decenas y unidades) y llamar a `digito_facil` con los símbolos adecuados.

Salida Esperada

Si el programa se ejecuta con el siguiente llamado:

```
digito_facil(7, 'X', 'V', 'I');
```

La salida debe ser:

```
VII
```

Solución del año pasado:

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

void digito_facil(int d, char h, char m, char l);
void imprime_romanos(int num);

int main() {
    digito_facil(7, 'X', 'V', 'I'); // Debe imprimir VII
    return 0;
}

void digito_facil(int d, char h, char m, char l) {
    switch (d) { // d va de 0 a 9
        case 0: break;
        case 1: cout << l; break;
        case 2: cout << l << l; break;
        case 3: cout << l << l << l; break;
        case 4: cout << l << m; break;
        case 5: cout << m; break;
        case 6: cout << m << l; break;
        case 7: cout << m << l << l; break;
        case 8: cout << m << l << l << l; break;
        case 9: cout << l << h; break;
    }
}

void imprime_romanos(int num) {
    // Extrae cada dígito y llama a digito_facil con los
       símbolos correctos
    digito_facil(num / 1000, ' ', ' ', 'M'); // Miles
    digito_facil((num / 100) % 10, 'M', 'D', 'C'); // Centenas
    digito_facil((num / 10) % 10, 'C', 'L', 'X'); // Decenas
    digito_facil(num % 10, 'X', 'V', 'I'); // Unidades
}
```

Solución de este curso:

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

void imprime_romanos (int rom);

int main() {
    int romano;
    cout<<"Introduce el número (0-399)"<<endl;
    cin>>romano;
    imprime_romanos(romano);
    return 0;
}

struct Digitos{
    const char* const cad;
    int val;
};

#include<vector>
using std::vector;
const vector<Digitos> rom_digitos { {"M",1000} , {"CM",900},
   {"D",500}, {"CD",400}, {"C",100}, {"XC",90}, {"L",50},
   {"XL",40}, {"X",10}, {"IX",9}, {"V",5}, {"IV",4}, {"I",1}};
   //se crea un vector que se inicializa con estructuras de
   Digitos
void imprime_romanos(int rom) {
    for (const auto&rd:rom_digitos)
    while (rd.val <= rom){
        rom -= rd.val;
        cout<<rd.cad;
    }
}
```

5. Sesión 5. Tipos de datos definidos por usuario.

5.1. alias.

Para ver más información sobre el uso de esta definición de datos, consultar el apartado 1.5.

```
#include <iostream>
using namespace std;
int main()
{
    typedef int entero;
    using entero2  = int;
    entero A  = 13;
    entero2 B  = 10;
    cout <<"El valor de A es "<<A<<" y el valor de B es
       "<<B<<endl;
    return 0;
}
```

5.2. enum.

Existen dos tipos principales de `enum` que proporcionan una forma de definir un conjunto de constantes simbólicas.

5.2.1. enum unscoped. Sin ámbito.

Siguen la estructura:

```
enum nombre:tipo_subyacente{enum1=cte1,enum2=cte3,...}variables;
```

Características:

- Los enumerados son constantes en su ámbito. Están al mismo nivel que la definición.
- Los valores son implícitamente convertibles a `int`. Las constantes y variables promocionan automáticamente al tipo de datos subyacente. Mirar ejemplo 2 y 3.
- No se permite la asignación directa de un `int`.
- El `nombre`, `tipo_subyacente`, la inicialización inicial y las variables son opcionales.

Ejemplo 1:

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

enum Color {ROJO, VERDE, AZUL}; // Definición del enum no
   escopado
```

```
int main() {
    Color miColor = ROJO; // Asignar un valor del enum

    if (miColor == ROJO)
        cout << "El color es rojo." << endl;

    // Los valores de un enum no escopado se pueden convertir
       implícitamente a int
    cout << "El valor de ROJO es: " << ROJO << endl;
    cout << "El valor de VERDE es: " << VERDE << endl;
    cout << "El valor de AZUL es: " << AZUL << endl;
    return 0;
}
```

Ejemplo 2:

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

enum : char { rojo = 'r', verde = 56 };
int main() {
    cout << rojo << " " << (int)verde << endl; // Imprime: r 56
    cout << rojo << " " << verde << endl; // Imprime: r 8
    return 0;
}
```

Ejemplo 3:

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

enum : int { rojo = 'r', verde = 56 };
int main() {
    cout << rojo << " " << verde << endl; // Imprime: 114 56
    return 0;
}
```

Ejemplo 4. Si existen dos elementos del mismo valor, se confunden.

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

enum Estado {ACTIVO = 1, INACTIVO = 0, SUSPENDIDO};
   //SUSPENDIDO tiene mismo valor que ACTIVO

int main() {
    Estado estadoUsuario = SUSPENDIDO;

    if (estadoUsuario == ACTIVO) // Comparación
        cout << "El usuario está ACTIVO." << endl;
```

```
    if (estadoUsuario == SUSPENDIDO) // Pero en realidad lo
       asignamos como SUSPENDIDO
        cout << "El usuario está SUSPENDIDO." << endl;
    return 0;
}
```

5.2.2. `enum scoped`. Con ámbito

Siguen la estructura:

```
enum class nombre:tipo_subyacente { , , , , , , } variables;
```

También permite el uso de `struct`.

```
enum struct nombre:tipo_subyacente { , , , , , , } variables;
```

Características:

- Los enumerados comparten ámbito con la enumeración. Inyecta los nombres con ámbito propio.
- No se convierten implícitamente a `int`. No hay conversión implícita al tipo de datos subyacente.
- No se permite la asignación directa de un `int`.
- Puedes especificar el tipo subyacente (como `int`, `char`, `unsigned`, etc.), aunque por defecto es `int`.
- Debes usar el nombre del `enum` para acceder a sus miembros.
- El único elemento opcional es el tipo_subyacente.

Ejemplo 1:

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

enum struct Color : int {RED = 1, GREEN, BLUE};
enum class Size : char {SMALL = 'S', MEDIUM = 'M', LARGE = 'L'};
int main() {
    Color c = Color::RED;
    // cout << c << endl; // Error: No hay conversión implícita
       a int

    cout << static_cast<int>(c) << endl; // Necesita cast
       explícito

    Size s = Size::SMALL;
    cout << static_cast<char>(s) << endl; // Salida: S

    return 0;
}
```

Ejemplo 2:

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

int main(){
    enum class Color { red, green = 20, blue };
    Color r = Color::green;

    switch (r)
    {
        case Color::red:
            cout << "Rojo";
            break;
        case Color::green:
            cout << "Verde";
            break;
        case Color::blue:
            cout << "Azul";
            break;
    }
    return 0;
}
```

Ejemplo 3:

```
#include <iostream> //Versión avanzada del ejemplo anterior
using namespace std;

int main() {
    enum class Color { red, green = 20, blue };
    Color r = Color::green;

    // Habilitar el uso de los enumeradores sin "Color::"
    using enum Color; //using enum es una característica de
       C++20!!

    switch (r) {
        case red:
            cout << "Rojo";
            break;
        case green:
            cout << "Verde";
            break;
        case blue:
            cout << "Azul";
            break;
    }
    return 0;
}
```

5.2.3. `static_cast`.

`static_cast` es un operador de conversión utilizado en C++ para realizar conversiones de tipos de datos en tiempo de compilación. Su principal función es convertir un valor de un tipo de datos a otro de manera segura y explícita. Puede ser utilizado para realizar conversiones de tipos que el compilador considera seguras y que no implican una pérdida de información, como convertir entre tipos numéricos compatibles, punteros y referencias.

Conversiones numéricas.

Convierte entre tipos de datos numéricos compatibles.

```
#include <iostream>

int main() {
    double numero_double = 3.14;
    int numero_entero = static_cast<int>(numero_double);

    std::cout << "Número double: " << numero_double <<
       std::endl;
    std::cout << "Número entero: " << numero_entero <<
       std::endl;
    return 0;
}
```

Conversión a Enumeraciones.

Convierte un valor entero a un tipo enumerado de forma explícita.

```
#include <iostream>
using namespace std;
enum Color { RED, GREEN, BLUE };

int main() {
    int n = 1;
    Color c = static_cast<Color>(n);
    cout << c << endl; // Imprime: 1 (valor interno del enum)
    return 0;
}
```

Conclusión.

`static_cast` es una herramienta fundamental en C++ para realizar conversiones de tipo seguras y claras en tiempo de compilación. Siempre que necesites realizar una conversión que sea lógica y no requiera verificación en tiempo de ejecución, es preferible utilizar `static_cast` en lugar del casting estilo C.

5.3. struct.

En C++, una estructura (definida con la palabra clave `struct`) es un tipo de dato compuesto que, conceptualmente, es equivalente a una clase pero con acceso público por defecto a sus miembros. Las estructuras pertenecen al paradigma de programación orientada a objetos y representan un caso particular de clase.

En el contexto de programación de sistemas y bajo nivel, se trabaja frecuentemente con estructuras POD (Plain Old Data), que son un tipo especial de estructura que:

- Solo contiene datos miembros convencionales (sin funciones miembro)
- No tiene constructores o destructores definidos por el usuario
- No utiliza herencia ni polimorfismo

Ejemplo:

```
#include <iostream>
using namespace std;
struct Punto2D {
    double x;
    double y;
};

struct Segmento2D {
    Punto2D p1, p2;
    char etiqueta[10];
};

int main() { // Uso de estructuras
    Punto2D pa{1.0, 2.0}, pb{2.0, 2.0}, pc{pb};
    Punto2D pd{}, *pp = &pa, *pp2{&pa};
    cout << pp->x<<' ';
    cout << (*pp).x;
    Segmento2D s1{pa, pb}, s2{pa};
    Segmento2D s3{{0, 1}, {1, 1}, "dos"}, *ps = &s3;
    s2 = {{1, 1}, {3, 4}, "tres"};
    s3 = {{1, 1}};
    pp = &(s2.p1);
    return 0;
}
```

Comentarios a realizar:

- La estructura `Punto2D` contiene dos campos `double`, uno para `x` y otro para `y`. También podría definirse en una sola línea como:

  ```
  double x, y;
  ```

- `Segmento2D` demuestra que una estructura puede contener otras estructuras como miembros. Aquí contiene dos puntos y una cadena de caracteres como etiqueta.

```
struct Segmento2D {
    Punto2D p1, p2;
    char etiqueta[10];
};
```

- La siguiente línea muestra cómo se pueden inicializar estructuras directamente y también por copia (como `pc` a partir de `pb`).

```
Punto2D pa{1.0, 2.0}, pb{2.0, 2.0}, pc{pb};
```

- Al omitir valores, los campos se inicializan a cero:

```
Punto2D pd{};
```

- Se declara un puntero a una estructura y se inicializa. `pp` es un puntero que apunta a pa y es equivalente a `pp2`.

```
Punto2D *pp = &pa, *pp2{&pa};
```

  Se permite el acceso a los miembros mediante `->` o desreferenciando con `*`. Ambas sentencias son equivalentes.

```
cout << pp->x;
cout << (*pp).x;
```

- En la siguiente línea, `s2` se inicializa parcialmente, por lo que `p2` y `etiqueta` se rellenan con ceros automáticamente.

```
Segmento2D s1{pa, pb}, s2{pa};
```

- Se muestra también cómo se puede usar la inicialización por listas en estructuras anidadas:

```
Segmento2D s3{{0, 1}, {1, 1}, "dos"};
```

- Es válida la asignación completa de estructuras:

```
s2 = {{1, 1}, {3, 4}, "tres"};
```

  y también se permite asignar solo parcialmente (lo no especificado se rellena con ceros o valores por defecto):

```
s3 = {{1, 1}};
```

- Finalmente, esta línea asigna a `pp` la dirección de uno de los campos internos de una estructura. Apunta a `p1` en `s2`.

```
pp = &(s2.p1);
```

- Notar que cada `double` ocupa 8 bytes, lo cual es relevante al hablar de memoria.

5.4. union.

En C++, una `union` es un tipo especial de estructura de datos que permite almacenar distintos tipos de variables en una misma región de memoria. A diferencia de las estructuras (`struct`), donde cada elemento ocupa su propia ubicación en memoria, en una `union` todos los miembros comparten exactamente el mismo espacio de almacenamiento. El tamaño total de la `union` corresponde al tamaño de su elemento más grande.

Es importante destacar que al asignar un valor a cualquier miembro de la `union`, se sobrescribirán automáticamente los valores de los demás miembros, ya que todos utilizan la misma posición de memoria de forma exclusiva.

Ejemplo 1:

```
union u { int a; const char *b; };
u a = {1};
u b = {2, "abcd"};
u c = {"abcd"};
u d = {.b = "abcd"};
```

Comentarios a realizar:

- Se define una `union` llamada `u`, que puede almacenar un `int` o un puntero a `const char`. Ambos miembros comparten el mismo espacio de memoria.
- En la siguiente línea se inicializa la unión con el valor `1` para el campo `a`. Las listas de inicialización en uniones solo inicializan **el primer miembro declarado**, a menos que se utilice sintaxis de C++20.

  ```
  u a = {1};
  ```

- La línea es inválida en C++. Aunque parece inicializar los dos campos, en realidad **solo puede inicializar uno** al usar lista de inicialización. Esto podría funcionar en C (no estándar), pero en C++ es un error.

  ```
  u b = {2, "abcd"};
  ```

- Igualmente esta línea también es problemática, ya que la inicialización por lista aplica al primer campo (`int a`), y no a `b`. Aquí el compilador intentará convertir `"abcd"` a un entero, lo cual no tiene sentido semántico.

  ```
  u c = {"abcd"};
  ```

- Desde C++20, es posible usar inicialización designada. Esta sintaxis permite inicializar explícitamente un campo distinto al primero, algo que antes no estaba permitido en C++.

  ```
  u d = {.b = "abcd"};
  ```

- Un aspecto fundamental del uso de `union` es la responsabilidad del programador.

Ejemplo 2:

```
#include <iostream>
using std::cout;
using std::endl;
using byte = unsigned char;

union U { int a; double b; };

//constexpression es una palabra nueva
constexpr int max_size() {
    return sizeof(U);
}

union Cosa {
    int numero;
    double real;
    byte bytes[max_size()];
};

int main() {
    Cosa mi_cosa{0x0015F130}, *p_cosa{&mi_cosa};
    for (auto c : mi_cosa.bytes)
        cout << static_cast<int>(c) << " ";
    cout << mi_cosa.real << endl;
    cout << std::hex << p_cosa->numero << endl;
    for (auto c : mi_cosa.bytes)
    cout << static_cast<int>(c) << " ";
    cout << endl << sizeof(mi_cosa) << endl;
    return 0;
}
```

Comentarios a realizar:

- Se utiliza `using byte = unsigned char;` para definir un alias de tipo. Esto es útil para acceder byte a byte a zonas de memoria.
- Se define una unión `U` con un `int` y un `double`. La función `constexpr max_size()` devuelve el tamaño de la unión, es decir, el mayor tamaño entre sus miembros.
- En `Cosa`, otra unión, se incluyen tres miembros:
  - `numero` de tipo `int`,
  - `real` de tipo `double`,
  - un array de `byte` cuyo tamaño es `max_size()` (típicamente 8 bytes).

  Todos estos miembros comparten la misma dirección de memoria debido a que están en una `union`.
- En la siguiente línea, se inicializa la unión con un valor entero. El valor se almacena en los primeros 4 bytes de la memoria que representa a la unión.

```
Cosa mi_cosa{0x0015F130};
```

- El siguiente bucle recorre byte a byte la memoria ocupada por la unión e imprime el contenido como números enteros. Al escribir un entero y leer como bytes, se revela la representación interna del número.

```
for (auto c : mi_cosa.bytes)
    cout << static_cast<int>(c) << " ";
```

- El acceso a `mi_cosa.real` puede producir comportamiento indefinido, ya que el valor real nunca fue escrito directamente. Estás leyendo los mismos bytes pero interpretándolos como un `double`.

```
cout << mi_cosa.real << endl;
```

- Se imprime el valor de `numero` en formato hexadecimal:

```
cout << std::hex << p_cosa->numero << endl;
```

- Finalmente, se imprime nuevamente el contenido byte a byte y luego el tamaño total de la unión:

```
cout << endl << sizeof(mi_cosa) << endl;
```

  lo cual debería ser 8 bytes (en la mayoría de arquitecturas) ya que es el mayor de los tamaños de los miembros (`int`, `double` o el arreglo de 8 bytes).

5.5. Enumeraciones, estructuras y uniones anónimas.

Una de las cosas que permite C++ es crear estructuras o agrupaciones anónimas. No hace falta poner nombre a la estructura.

```
void main(){
    struct {
        double real, imag;
    } C1; // C1 no es de ningún tipo nominal, no tiene nombre
    // es una cosa que solamente se usará dentro de la función
    C1.real = 8.0;
}

{
    union {
        double real;
        int entero;
    } var; // se tiene una variable que tiene un real y un
       entero
    // sin necesidad de crear más variables de ese tipo
    var.real = 8.0;
}

void mi_func(){
    union {
        int entero;
        double real;
    }; // en mi_funcion estoy creando o un entero o un real
    entero = 3;
    cout << real;
}
```

Comentarios a realizar:

- En la primera parte del código:

  ```
  struct {
      double real, imag;
  } C1;
  ```

  se define una estructura anónima dentro de la función `main`, es decir, no tiene un nombre de tipo, por lo que no puede ser reutilizada fuera de la función. Esta estructura es útil cuando solo se necesita un grupo de datos dentro de una función sin la necesidad de definir un tipo completo.

- La inicialización de `C1` se hace en la siguiente línea:

  ```
  C1.real = 8.0;
  ```

  En este caso, solo se utiliza el campo `real` de la estructura anónima, ya que no se necesita acceder al campo `imag`.

- A continuación, se presenta una **unión anónima**:

```
union {
    double real;
    int entero;
} var;
```

  La unión `var` puede almacenar un `double` o un `int`, pero solo uno de esos valores puede estar activo en cualquier momento. La estructura de la unión se define sin un nombre de tipo, lo que significa que la unión solo puede ser utilizada en ese bloque de código.

- Se asigna el valor `8.0` al miembro `real`:

```
var.real = 8.0;
```

  Esto modifica la parte de la memoria ocupada por `var`, y el campo `entero` no se debe usar, ya que está en la misma ubicación de memoria que `real`.

- En la función `mi_func`, se utiliza una **unión anónima**:

```
union {
    int entero;
    double real;
};
```

  Se define una unión sin nombre, lo que permite almacenar un `entero` o un `real` en el mismo espacio de memoria.

- Posteriormente, se asigna un valor a `entero`:

```
entero = 3;
```

  y se imprime el valor de `real`:

```
cout << real;
```

  En este caso, al imprimir `real`, se obtiene un valor indeterminado, ya que `real` y `entero` comparten la misma ubicación de memoria y solo uno de ellos debe ser usado a la vez. Esto puede llevar a comportamiento indefinido si se accede a la memoria de manera incorrecta.

- La responsabilidad del programador es clave cuando se usan estructuras o uniones anónimas, ya que el compilador no puede garantizar cuál miembro de la unión está activo en un momento dado. El programador debe gestionar cuidadosamente qué campo utilizar en cada situación.

Ejemplo de aplicación:

```
// Un ejemplo de las aplicaciones de estas estructuras
#include <iostream>

struct Data {  // este es un ejemplo de tipo variant
    enum : char { CARACTER, ENTERO, REAL } tipo;
    union {  // las uniones anonimas sirven dentro de
       estructuras para zonas alternativas
        char caracter;
        int entero;
        double real;
    };  // no tiene nombre
};

using std::cin, std::cout, std::endl;

void imprime(Data d) {
    switch(d.tipo) {
        case Data::CARACTER: cout << "Char:" << d.caracter <<
           endl;
        break;
        case Data::ENTERO: cout << "Int:" << d.entero << endl;
        break;
        case Data::REAL: cout << "Real:" << d.real << endl;
        break;
    }
}

int main() {
    Data mi_data{};
    short opcion;  // si yo hubiera puesto char opcion
    // en los switch debería haber puesto: case '1'
    cout << "1-char\n2-int\n3-real\n";
    cin >> opcion;

    switch(opcion) {
        case 1:
            mi_data.tipo = Data::CARACTER;
            // dentro de la clase data, la constante CARACTER
            cin >> mi_data.caracter;
            break;
        case 2:
            mi_data.tipo = Data::ENTERO;
            cin >> mi_data.entero;
            break;
        case 3:
            mi_data.tipo = Data::REAL;
            cin >> mi_data.real;
            break;
```

```
    }
    imprime(mi_data);
    // no hace falta poner el return 0, el sistema ya lo hace
}
```

5.6. Punteros relativos.

En C++, los operadores de acceso a miembro se utilizan para acceder a los miembros de una clase o una estructura.

Operadores acceso a miembro:

- `.`
- `->`

Operadores acceso a miembro a través de puntero a miembro:

- `.*`
- `->*`

Ejemplo 1:

```
struct Punto{
    int x,y,z;
}

Punto p1{1,2,3}, p2{4,5,6}, p3{7,8,9};

Punto *pd=&p1;//puntero a p1
pd -> y ;//accediendo a la direccion de y
(*pd).y; //accediendo al contenido
p_miembro = &Punto::y;//del tipo de datos Punto, la direccion y
p1.y = 17;//estas 3 siguientes expresiones son equivalentes
pd -> y = 17;
p1.*p_miembro = 17;//accede a la parte y de p1
p2.*p_miembro = 18;//equivale a p2.y
p_miembro = &Punto::z;
p2.*p_miembro = 18;//equivale a p2.z
//cabe destacar que los punteros relativos son como un offset,
   estamos indicando una
//diferencia y es por ello que nos podemos mover entre las
   distintas variables de Punto
```

Comentarios a realizar:

- Se crean 3 puntos y estos están dispuestos de manera ordenada en la memoria, siendo p1 el primero.

  ```
  Punto p1{1,2,3}, p2{4,5,6}, p3{7,8,9};
  ```

- Para obtener la dirección relativo, el puntero relativo tiene la forma:

  ```
  <tipo de datos apuntado> <puntero> <identificador>;
  ```

  Ejemplo:

  ```
  int Punto::* p_miembro; // donde Punto::* es una clase::*
  ```

Ejemplo 2:

```
struct Punto{
    int x,y,z;
}

Punto p{1,2,3}, *pp{&p};
double *pd{&p.y}, Punto::*prd{&Punto::y};

p.y;
```

Comentarios a realizar:

- Puntero relativo a entero. Es recomendable ver `Punto::*` como un pack.

  ```
  double *pd{&p.y}, Punto::*prd{&Punto::y};
  ```

- La siguiente línea accede a la variable y del tipo Punto.

  ```
  p.y;
  ```

  Es equivalente a todas las siguientes:

  ```
  pp->y;   (*pp).y;   *pd;   p.*prd;   pp->*prd;
  ```

Ejemplo 3:

```
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

struct Persona {
    char nombre[20];
    double edad;
    double alto;
    double cintura;
    double peso;
};

double media(const vector<Persona>&, double Persona::*v);

int main() {
    vector<Persona> pueblo{
        {"Pepe", 56, 170, 90, 77},
        {"Jorge", 45, 174, 94, 81},
        {"Luis", 77, 167.3, 88, 82.3},
        {"Carlos", 65, 175, 81, 73}
    };
    cout<<"Edad media: "<<media(pueblo,&Persona::edad) <<endl;
    cout<<"Peso medio: "<<media(pueblo,&Persona::peso) <<endl;
    cout<<"Altura media: "<<media(pueblo,&Persona::alto) <<endl;
    cout<<"Cintura media: "<<media(pueblo,&Persona::cintura)
       <<endl;

    return 0;
}

double media(const vector<Persona>& p, double Persona::*v) {
    double media = 0;
    for (const auto& i : p) {
        media += i.*v;
    }
    return media / p.size();
}
```

Comentarios a realizar:

- Creamos función para calcular la media de cualquiera de los campos de Persona. Observar el uso del operador `&` , recordar su uso para objetos grandes en el apartado 4.3.1.

  ```
  double media(const vector<Persona>&, double Persona::*v);
  ```

  Pasa el vector original con el compromiso de que no lo vas a modificar.

- Definición de la función.

```
double media(const vector<Persona>& p, double Persona::*v){
}
```

Donde:

```
double Persona::*v //puntero relativo a double
```

6. Sesión 6. Referencias y funciones.

6.1. Referencias.

La mejor manera de entender el concepto de referencia es pensar que es un alias, un nombre alternativo para un objeto. No ocupan memoria. Emplea el operador `&` y este alias va a seguir las mismas reglas que una variable, pero puede hacer referencia a un objeto.

La potencia de las referencias surge cuando se emplea en las funciones, ya que nos permiten pasar argumentos por referencia y modificar el valor original de la variable.

Todas las referencias deben estar inicializadas. Maneras equivalentes de inicializar:

```
<tipo> &<identif> = <variable de ese tipo>
<tipo> &<identif> {variable de ese tipo}
```

Ejemplo 1:

```
int a,b;
int &c = a; //c referencia  a la variable 'a'
int &d{a}, &e{c}, *h{&a}, f, g[3]; //se puede escribir como:
int &d{a}, &e{a}, *h{&c}, f, g[3];
```

Ejemplo 2:

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

int main() {
    int a = 2, &b{a};
    cout<<a<<" "<<b<<endl;
    a = 4;
    cout<<a<<" "<<b<<endl;
    int i, &j{i};
    for(i=0; j<4;i++)
        cout <<j;
    return 0;
}
```

Las referencias salvo las que son parámetros o valores de retorno de una función, debe de ser explícitamente referenciadas.

Las referencias no pueden ser reasignadas una vez realizadas.

Las referencias no son objetos.

- No existen arrays de referencia.
- No existen punteros a referencias.
- No exsiten referencias a referencias.
- No existen referencias a void.

Ejemplo 3:

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

int i=0;//i1
int main()
{
    int i=0,&j{i};//esta i no es la primera, es i2
    while(::i<3){ //i1
        ::i++;//i1
        i=0;//i2
        for(int i=0;i<3;i++)//i3 i3 i3
            cout<<::i<<j<<i<<endl;//i1 i2 i3
    }
    return 0;
}
```

Cuestión. Significado de las siguientes sentencias:

```
float a, *p{&a}; //puntero p a float inicializado con la
   dirección de a
float * &v = p; //v es una referencia a un puntero p
float d[3], (&j)[3]{d}; //j es una referencia a un vector de 3
   floats
```

6.1.1. Ejemplo básico. Función swap en C++.

Nos permite intercambiar dos variables.

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

void swap(int &a, int &b) { //regla de oro?
    int c = a;
    a = b;
    b = c;
}

//Ejemplo de uso
int main() {
    int x = 3, y = 4;
    swap(x,y);
    cout<<x<<' '<<y;
    return 0;
}
```

Siguiente:

```
int d;
const int &a = 5;
int &&b = 5; //rvalue, permitido
int &&b = d; //esto no se admite, una referencia a un rvalue
   con una variable
const int &a = d;//una referencia constante sí puede ser
   inicializada con una variable
//utilidad? evitar tener que hacer una copia

//ejemplo
struct GRANDE{
    char nombre[50];
    char apellidos[200];
    int edad;
    double peso;
    char direccion[1000];
};

void imprime_dato(const GRANDE &p){ //ventaja: p puede ejecutar
   los métodos const

}
```

Ejercicio propuesto, algoritmo de ordenación por el método de la burbuja. Siendo el main:

```
int main() {
    vector<int> v1(10), v2, v3;
    for (auto &num : v1) num = random(100);
    v2 = v3 = v1;
    ordenar_bb(v1);
    ordenar_sd(v2);
    sort(begin(v3), end(v3));
    imprime(v1);
    imprime(v2);
    imrpime(v3);
    return 0;
}
```

Solución:

```
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int random(int max) {
    static bool init = (srand(time(NULL)), true);
    return rand() % max;
}

void imprime(const vector<int> &v) {
    for (const auto& num : v)
        cout << num << " ";
    cout << endl;
}

void ordenar_bb(vector<int>&v) {
    // Con el for de rango, no sabemos donde están las cosas,
       luego no se recomienda
    // Se recomienda usar índices
    bool cont = true;
    int n = v.size();
    while ((n > 0) && cont) {
        n--;
        cont = false;
        for (int i = 0; i < n; i++) {
            if (v[i] > v[i + 1]) {
                swap(v[i], v[i + 1]);
                cont = true;
            }
        }
    }
}

void ordenar_sd(vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        for (int j = i + 1; j < v.size(); j++) {
            if (v[i] > v[j])
            swap(v[i], v[j]);
        }
    }
}

int main() {
    vector<int> v1(10), v2, v3;
    for (auto &num : v1) num = random(100);
```

```
    v2 = v3 = v1;
    ordenar_bb(v1);
    ordenar_sd(v2);
    sort(begin(v3), end(v3));
    imprime(v1);
    imprime(v2);
    imprime(v3);
    return 0;
}
```

6.1.2. Lvalue.

Un `lvalue` (locator value) es algo que tiene una dirección de memoria. Es decir, puedes tomar la dirección con `&`. Tiene una serie de características:

- Objetos persistentes???
- Tiene nombre (o entidad).
- Puede estar en el lado izquierdo de una asignación.
- Ocupa un espacio de memoria y se puede modificar (si no es `const`).

Ejemplo:

```
int x = 10;     // 'x' es un lvalue
int* p = &x;    // válido, porque x tiene dirección
```

Ejemplo: la referencia como valor de retorno.

```
v.at(3) = 8;
v[3] = 8;
```

6.1.3. Rvalue.

Un `rvale` (read value) es un valor temporal, sin nombre, que no puedes tomarle la dirección. Tiene una serie de características:

- Son típicamente temporales o literales.
- No viven más allá de la expresión en la que se usan.
- No puedes asignales otro valor directamente.

Ejemplo:

```
int y = 5;       // el literal '5' es un rvalue
int z = x + y;   // la expresión 'x + y' es un rvalue
```

Ejemplo práctico.

```
int x = 10;     // 'x' es un lvalue
int y = x;      // 'x' es lvalue, 'y' es lvalue, pero 'x'
   también actúa como rvalue en el lado derecho
int z = x + y; // 'x + y' es un rvalue (no tiene nombre, no
   vive mucho tiempo)

int* p = &x;    // válido
// int* q = &(x + y); // error: no se puede tomar dirección de
   un rvalue
```

6.1.4. La referencia como valor de retorno.

La idea es que si una función retorna una referencia, está retornando una variable de alguna manera.

```
v.at(3) = 8; // esta es la expresión que nosotros buscamos
   entender mejor
v[3] = 8;
```

Ejemplo:

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

struct Punto {
    int x, y, z;
};

int &menor(Punto &p) {
    return p.x < p.y ? (p.x < p.z ? p.x : p.z) : (p.y < p.z ?
       p.y : p.z);
}

int main() {
    Punto p[6]{{1, 2, 3}, {2, 3, 1}, {3, 1, 2}, {2, 1, 3}, {3,
       2, 1}, {1, 3, 2}}, p2{3, 4, 1};
    for (auto &e : p) {
        cout << menor(e) << ' ';
        menor(e) = 4;
        cout << menor(e) << ' ';
        menor(e) = 5;
        cout << menor(e) << endl;
    }
    cout<<p[0].x<<p[0].y<<p[0].z;
}
```

Comentarios a realizar:

- La ejecución de la función menor no devuelve un valor. Devuelve una referencia a la posición de la coordenada más pequeña de un punto.

```
int &menor(Punto &p) {
    return p.x < p.y ? (p.x < p.z ? p.x : p.z) : (p.y < p.z
       ? p.y : p.z);
}
```

- Se emplea el operador ternario para intentar reversar un poco la función.

```
return p.x < p.y ? (p.x < p.z ? p.x : p.z) : (p.y < p.z ?
   p.y : p.z);
```

- P es un vector de seis puntos.

```
Punto p[6]{{1, 2, 3}, {2, 3, 1}, {3, 1, 2}, {2, 1, 3}, {3,
   2, 1}, {1, 3, 2}}, p2{3, 4, 1};
```

- Iterando con la referencia e al punto p, permite modificarlo al iterar.

```
for (auto &e : p) {
    cout << menor(e) << ' '; //identifica el menor
    menor(e) = 4; //cambia su valor por 4
    cout << menor(e) << ' '; //identifica el siguiente menor
    menor(e) = 5; //cambia su valor por 5
    cout << menor(e) << endl;
}
```

- Imprimir elementos del primer punto:

```
cout<<p[0].x<<p[0].y<<p[0].z;
```

6.1.5. Ejercicio de examen.

Pregunta: ¿Qué imprime?

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

int &foo(vector<int> &v) {
    int j = 0;
    for (int i = 0; i < v.size(); i++)
        if (v[i] < v[j]) j = i;
    return v[j];
}

int main() {
    vector<int> lista(3);
    for (auto n : {12, 13, 14, 15, 16, 17, 18, 12, 13, 14})
        foo(lista) = n;
    for (auto n : lista)
        cout << n << " ";
    return 0;
}
```

Solución:

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

// Foo devuelve una referencia al menor elemento del vector
int &foo(vector<int> &v) {
    int j = 0;
    for (int i = 0; i < v.size(); i++)
        if (v[i] < v[j]) j = i;
    return v[j];
}

int main() {
    vector<int> lista(3); // Vector de 3 elementos
       inicializados en cero
    for (auto n : {12, 13, 14, 15, 16, 17, 18, 12, 13, 14})
        foo(lista) = n; // Asigna 'n' al menor elemento actual
           del vector
    for (auto n : lista)
        cout << n << " ";
    return 0;
}
```

Imprime:

```
18 14 17
```

6.2. Funciones.

6.2.1. Sobrecarga de funciones.

Sobrecargar funciones quiere decir que el significado de las funciones va a cambiar en base al argumento al que se le aplique.

Dos funciones se pueden llamar igual siempre que se puedan diferenciar en el tipo, número y orden de sus argumentos. No se diferencian en base a sus valores de retorno.

Ejemplo 1:

```
#include <iostream>
using namespace std;
struct Complex {
    double re, im;
};

Complex suma(Complex a, Complex b){// suma 1
    Complex e{a.re + b.re, a.im + b.im};
    return e;
}

double suma(double x, double y){// suma 2
    return x + y;
}

double suma(double x){// suma 3
    return x + 1.0;
}

int main(){
    Complex c1{1, 1.5}, c2{0, 1.1};
    auto r1 = suma(c1, c2);    // usa la suma 1
    auto r2 = suma(3.4, 2.3);  // usa la suma 2
    auto r3 = suma(3, 4);      // usa la suma 2, convierte el
       entero a double
    auto r4 = suma(2);         // usa la suma 3, convierte int
       a double
    auto r5 = suma(3.4, 5);    // usa la suma 2
    auto r6 = suma(3.0F, 5.3); // usa la suma 2, convierte
       float a double

    cout << "r1: " << r1.re << " + " << r1.im << "i" << endl;
    cout << "r2: " << r2 << endl;
    cout << "r3: " << r3 << endl;
    cout << "r4: " << r4 << endl;
    cout << "r5: " << r5 << endl;
    cout << "r6: " << r6 << endl;
    return 0;
}
```

Si se añade la función de suma 4.

```
#include <iostream>
using namespace std;
struct Complex {
    double re, im;
};

Complex suma(Complex a, Complex b){// suma 1
    Complex e{a.re + b.re, a.im + b.im};
    return e;
}

double suma(double x, double y){// suma 2
    return x + y;
}

double suma(double x){// suma 3
    return x + 1.0;
}

double suma(int a, int b){// suma 4
    return a + b;
}

int main(){
    Complex c1{1, 1.5}, c2{0, 1.1};
    auto r1 = suma(c1, c2);    // usa la suma 1
    auto r2 = suma(3.4, 2.3);  // usa la suma 2
    auto r3 = suma(3, 4);      // ahora usa la suma 4
    auto r4 = suma(2);         // usa la suma 3, convierte int
       a double
    //  auto r5 = suma(3.4, 5);    // Error de ambigüedad:
       puede ser suma 2 (double, double) o suma 4 (int, int)
    auto r6 = suma(3.0F, 5.3); // usa la suma 2, convierte
       float a double

    cout << "r1: " << r1.re << " + " << r1.im << "i" << endl;
    cout << "r2: " << r2 << endl;
    cout << "r3: " << r3 << endl;
    cout << "r4: " << r4 << endl;
    cout << "r6: " << r6 << endl;
    return 0;
}
```

6.2.2. Parámetros por defecto.

Se puede asumir que si un parámetro no es especificado por el usuario, entonces tendrá un parámetro por defecto.

Regla de oro: los parámetros por defecto deben de estar en el prototipo (la declaración), no en la definición.

Ejemplo:

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

// Declaración de la función con parámetro por defecto
void cuenta(int num, int ini = 1);  //ini tiene valor por
   defecto 1

int main() {
    // Ejemplos de uso:
    cuenta(3, 4); //Imprime: 4 5 6
    cuenta(3, 1); //Imprime: 1 2 3 (equivalente a cuenta(3))
    cuenta(3);    //Imprime: 1 2 3 (usa ini=1)
    // cuenta( ,3); //Inválido
    return 0;
}

// Definición de la función
void cuenta(int num, int ini) {// Nota: No se repite '=1' en la
   implementación
    for (int i = 0; i < num; i++) {
        cout << i + ini << " ";
    }
    cout << endl;  // Añadido para mejor formato
}
```

Cuestión: ¿Porqué la siguiente línea del ejemplo anterior es inválida?

```
cuenta( ,3);
```

Porque no puedes omitir un argumento en medio de la lista de parámetros.

Veamos un ejemplo adicional para responder esta cuestión. Si se tiene la función genérica foo:

```
int foo(int x=3, int y=2, int z=3){
    return x + y + z;
}
```

Cuestión: ¿Cuáles son válidas?

```
foo();          //válido
foo(2,3);       //válido
fooo( , ,3);    //error
```

6.2.3. Funciones `inline`.

Las funciones significan `inline` significan literalmente que esas funciones van a pedir al compilador que, si puede, sustituya la llamada a la función por su código.

1. Va asociada a la definición de la función, no a la declaración.

```
inline<definición de la función>;
```

2. Para hacer la sustitución del código, es necesario conocer el cuerpo de la función. Por lo tanto, la definición tiene que estar antes del uso de la función. Esta se suele colocar en los ficheros.h para así no tener errores.

3. `inline` es solo una sugerencia, no una orden. El compilador es capaz de tomar una decisión en base a rendimiento u otros factores, luego, puede ser ignorada.

4. En las clases, cuando se define un método, todas las funciones que estén definidas en el cuerpo de la clase se consideran `inline` por defecto.

5. Si yo consigo que una función sea `inline`, el programa es más rápido

¿Cuándo compensa definir una función `inline`? Cuando el proceso de llamada tiene un coste igual o mayor que la operación que se está realizando.

- Funciones que se usan muchas veces pero aparecen en pocos sitios.
- Funciones de cuerpo muy pequeño.

Ejemplo de la cuarta idea. Estructura típica:

```
struct Foo { //foo quiere decir una clase cualquiera
    // etc
    int haz_algo();
};

inline int Foo::haz_algo(){
    // etc
}
```

Consultar ejercicio de examen: 6.1.5.

7. Sesión 7. Dynamic memory.

7.1. Heap.

Ideas clave:

- Se caracteriza por ser memoria libre/`dynamic`.
- Caben objeto muy grandes y nos permite destruir objetos en cualquier orden.
- Permite liberar memoria a petición.
- No sigue una disposición ordenada (fragmentación) pero es persistente.
- En C/C++ se accede a través de las funciones:

```
C: malloc
C++: new
```

7.2. Stack.

Ideas clave:

- Se caracteriza por ser memoria `automatic`.
- Es una memoria pequeña, muy rápida y ordenada.
- La idea de esta memoria es que no puedo destruir algo sin destruir lo que está arriba. Es decir, el acceso no es aleatorio y el orden de construcción determina el orden de destrucción.

7.3. Ejemplo.

```
//Ejemplo, tenemos un bloque:
{
    int a = 3;
    vector<int> m(3,1);          /*(1)*/
    {                            /*(2)*/
        int d = 8;
        vector<int> v2 = m;      /*(3)*/
        foo(v2, d);              /*(4)*/
        m.push_back(8);          /*(5)*/
    }
} //se destruye el vector m

void foo(vector<int> v, int x){ /*(6)*/
    //etc
}
```

Comentarios a realizar:

- Línea (1). Se define `m` es que es una variable automática
- Línea (2). Al introducir unas llave, se abre otro contexto.

```
{
    int d = 8;
    vector<int> v2 = m;
    foo(v2, d);
    m.push_back(8);
} //se destruye d y v2 al terminar el contexto
```

- Línea (3). Se crea espacio en Heap y ocupa lo mismo que cualquier otro vector. Se copia la información que hay en `m` en `v2`.
- Línea (4). Llamada a foo.
- Línea (5). Se va a reservar espacio nuevo en Heap, donde se copia la info. La memoria Heap que teníamos reservada previamente, no se elimina, se traslada a la nueva ampliada.
- Línea (6). Definición de la función. Al terminar de ejecutarse, se borran los datos asociados.

7.4. New.

Intenta reservar espacio y construir e inicializar en este espacio un objeto o arreglo (vector) de objetos. El retorno es un puntero al objeto o al primer objeto del array (vector de objetos). Si falla nos devolverá un puntero a `nullptr` y lanza una excepción.

Existen 4 sintaxis disponibles:

- Para un único objeto:

```
new <tipo><inicializadores del objeto>;
new (tipo)<inicializadores del objeto>;
```

- Para un número `#` de objetos:

```
new tipo[#]<inicializadores de lista>;
new (tipo[#])<inicializadores de lista>;
```

7.4.1. Inicialización de un objeto.

a) No se pone nada → inicialización por defecto.

```
int *p = new int; /*equivalente a*/ auto p = new int;
```

b) Con paréntesis → construcción directa.

```
int *q = new int(3);
new int(3);
auto v = new vector<int>(3,2); /*equivalente a*/ auto v = new
    (vector<int>)(3,2);
```

c) Si ponemos {} se inicializa como lista de agregados.

```
auto v = new vector<int>{3,2,4});
```

7.4.2. Inicialización de N objetos, arrays.

a) No se pone nada → inicialización por defecto de cada elemento.

```
int *p = new int[10];//quiero crear 10 enteros de tipo int
```

b) Si ponemos ( ) paréntesis vacíos, cada elemento es inicializado por defecto como una {}.

```
int *p = new int[10](); /*equivalente a*/ int *p = new
    int[10]{};
```

c) Si ponemos {...} entonces es inicialización por lista.

```
int *p = new int[10]{1,2,3}; //se inicializan los 3 primeros
    elementos, el resto ceros
char *q = new char[10]{"hola"};
```

7.4.3. Delete.

Reglas de oro:

1. Siempre que hay un `new`, tiene que tener un `delete`.
2. Siempre que llamamos a `delete` desde un `nullptr` no pasa nada.
3. **Delete llama al destructor.**
4. **Todo lo que se crea dinámicamente se debe eliminar dinámicamente.**
5. **Lo que se reserve sin `[ ]` se libera sin `[ ]`.**

7.4.4. Ejemplos de borrado.

```
#include <iostream>;
int *a, *b, *c;

void(){
    a = new int;
    b = new (int);
    auto c = new int[3];
    int num = 8;
```

```
    auto f = new int[num];
    int g[num];
    int *g = new int[5]{1,2,3};
    int *i = new int[5]();
    int *j = new int[5]();
    int(*k)[3] = new int[8][3];
    int *l[8] = {new int[3], new int[3], new int[3]};
}
```

Comentarios a realizar para el borrado:

```
a = new int; //delete a;
b = new (int); //delete b;
auto c = new int[3]; //delete[] c; borra un array apuntado por c

int num = 8;
auto f = new int[num]; //delete[] f;

int g[num]; //prohibido, el tamaño del vector debe ser una
variable constante
int *g = new int[5]{1,2,3}; //delete[] g;
int *i = new int[5](); //delete[] i;
int(*k)[3] = new int[8][3]; //puntero a array de 3 enteros, 8
filas
int *l[8] = {new int[3], new int[3], new int[3]};
```

Comentarios a realizar:

- Reserva 8 grupos de enteros.

  ```
  int(*k)[3] = new int[8][3];
  ```

  Entonces, este número puede ser variable pero siempre son grupos de 3 enteros. Es por ello que el puntero tiene la forma `int(*k)[3]`. El modo de borrado es exactamente igual al ser un vector.

  ```
  delete[] k;
  ```

- Arreglo de 8 punteros a `int`, los tres primeros inicializados con arrays de 3 enteros y el resto vacíos. Ojo porque el tamaño de los vectores que se inicializan pueden variar.

  ```
  int *l[8] = {new int[3], new int[3], new int[3]};
  ```

  Para borrar, se tiene que ir uno a uno solamente en los inicializados.

  ```
  // delete[] l[0];
  // delete[] l[1];
  // delete[] l[2];
  // Nota: l[3]...l[7] quedan sin inicializar, cuidado con
     delete ahí
  ```

También se podría realizar un bucle `for` para el borrado. Sin embargo, se deben de inicializar todos los elementos.

```
int *l[8] = {new int[3], new int[3], new int[3], nullptr,
   nullptr, nullptr, nullptr, nullptr};
for (auto p : l) {
    if (p) delete[] p;
}
```

7.4.5. Ejemplo.

```
#include<iostream>
#include<cstring>
constexpr int TRAMO=5;
using namespace std;
int main()
{
    int tam = 0, cap = TRAMO, valor;
    auto vector = new int[TRAMO]{};//el {} es opcional
    while(cin>>valor, valor!=0)
    {
        vector[tam++] = valor;
        if(tam == cap)
        {
            cap+=TRAMO;
            auto aux = new int[cap];
            memcpy(aux,vector,tam*sizeof(int));
            delete []vector;
            vector = aux;
        }
    }
    cout<<"Se han introducido "<<tam<<" numeros"<<endl;
    for(int i=0; i < tam; i++) cout<<vector[i]<<endl;
    delete[]vector;
    return 0;
}
```

8. Sesión 8. Programación Orientada a Objetos.

8.1. Introducción.

Un objeto es una instancia de una clase y es una entidad que engloba una serie de datos y funciones. Es decir, un objeto queda definido y descrito por sus atributos y sus métodos.

El punto de partida fundamental es visualizar desde el inicio claramente las funcionalidades/rol que deseamos que una clase tenga. En caso contrario, es un error fatal.

El fundamento que reside detrás de la creación de clases es crear nuestro propio tipo de datos básico. Para ello, tendremos que plantearnos:

1. Cómo copiar cualidades entre individuos.
2. Cómo asignar las cualidades a los individuos.
3. Cómo se crean.
4. Cómo se destruyen.
5. Cómo operar entre ellos.

Un objetivo adicional, es contar con mecanismos de protección en las clases. Para lograr esto, va a ser necesario contar con una idea que son los niveles de acceso. Es decir, va a haber segmentos públicos, privados o protegidos.

- Público: Este nivel de acceso permite que cualquier parte del código, sin restricciones, pueda interactuar con los atributos y métodos de la clase.
- Privado: Este nivel restringe el acceso únicamente a los métodos de la propia clase. Los atributos y métodos privados están completamente ocultos para las clases externas y sólo pueden ser manipulados desde dentro de la clase donde fueron definidos.
- Protegido: Similar al nivel privado, restringe el acceso a la propia clase y también a las clases que heredan de ella.

Las clases tendrán:

- Datos miembro: llamados atributos que pueden ser 0 o infinitos. Diferencian los objetos.
- Funciones miembro: llamadas métodos que pueden ser 0 o infinitas. Comunes a todos los objetos de una clase.
- Niveles de acceso: `public`, `private` o `protected`.

Finalmente, en base a lo comentado, establecemos una serie de normas generales:

- Cada clase tiene que tener un propósito/rol.
- Hacer todos los datos privados por defecto y se usan métodos de interfaz para acceder a estos datos.
- Cuando las clases sean triviales y su información que contengan sean libres, entonces sus datos no tienen que ser privados, se hacen públicos.
- Incluir el calificativo `const` para todos los métodos que no modifiquen al propio objeto.
- Si una funcionalidad no requiere acceder a los datos privados del objeto o solo necesita utilizar su interfaz pública, lo normal es que sea una función externa a la clase. En caso contrario, es método.
- Poner nombres descriptivos y claros en las interfaces y funciones.

8.2. Ejemplo básico. Contador

Se propone crear un elemento contador, un objeto que es capaz de contar solo de 1 en 1 en sentido incremental. Que se puede inicializar en cualquier valor siendo positivo, y por defecto inicia en 0.

Ideas clave:

1. Los miembros de una clase están vinculados al ciclo de vida del objeto que los contiene. Su duración comienza cuando se crea el objeto y termina cuando este se destruye.

```
struct Contador {
    int cuenta; // inicializada a 0

    // Función que añade un 1 a la cuenta
    void incrementa() {
        cuenta++;
    }
};
```

2. El orden de la declaración de los miembros dentro de una clase es relevante para su inicialización. Sin embargo, para el uso dentro de la clase donde solamente se declaran, es irrelevante.

```
struct Contador {
    // Función que añade un 1 a la cuenta
    void incrementa() {
        cuenta++;
    }

    int cuenta; // inicializada a 0
};
```

   Se continúa con el ejemplo:

```
#include <iostream>
using namespace std;
struct Contador {
    int cuenta;
    // Función que añade un 1 a la cuenta
    void incrementa() {
        cuenta++;
    }
};

int main() {
    Contador micontador{}, micontador2{3};  // Inicializo
       micontador a 0
    // La cuenta de micontador2 es 3
    cout << micontador.cuenta << endl;
```

```
    micontador.incrementa(); // Ejecuta la función
       incrementa
    // Los métodos por defecto reciben todos los datos del
       objeto invocador. Por tanto, en este caso, son los
       datos de micontador.

    cout << micontador.cuenta << endl; // Imprime valor

    micontador2.incrementa(); // Se ejecuta la función con
       micontador2
    cout << micontador2.cuenta << endl;
    return 0;
}
```

3. Introducir niveles de protección es importante en las clases. En este caso, este código no funciona porque al privatizar los datos y métodos de la estructura Contador, no podemos usarlos en el main.

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

struct Contador {
    private:
    int cuenta;
    void incrementa() {
        cuenta++;
    }
};

int main() {
    Contador micontador{};
    cout << micontador.cuenta << endl; //error
    micontador.incrementa(); //error
    cout << micontador.cuenta << endl; //error
    return 0;
}
```

4. Las funciones miembro tienen derecho de acceso directo a las variables miembro y a las demás funciones miembro. Luego, la solución que planteamos es añadir un método adicional público que nos ayude a acceder a la variable privada cuenta.

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

struct Contador {
    private:
    int cuenta;
```

```
    public:
    void incrementa(){
        cuenta++;
        cout<<valor()<<endl; //A pesar de que la funcion
           valor esta definida a posterior, no da error
           porque puede acceder a la función miembro.
    }
    int valor(){
        return cuenta;
    }
};

int main() {
    Contador micontador{};
    cout << micontador.valor() << endl; //compila
    micontador.incrementa(); //compila
    cout << micontador.valor()<< endl; //compila
    return 0;
}
```

5. Una estructura de carácter privada es equivalente a una clase. Luego:

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

class Contador{
    int cuenta;
    public:
    void incrementa(){
        cuenta++;
        cout<<valor()<<endl;
    }
    int valor(){
        return cuenta;
    }

    //se tienen dos contadores: c1 y c2. Cada uno
       incrementa de una manera, y hacemos que
    //estoy accediendo a la parte privada de un objeto del
       mismo tipo que yo
    void haz_igual(Contador *otro){ //recibe un contador
        otro->cuenta = cuenta; //modifico la info del
           contador recibido
    }
};
```

Implementación completa en ficheros.

Contador.h:

```
class Contador
{
    int cuenta = 0;
    public:
    void incrementa(); //si modifica el objeto
    int valor const () //no modifica el objeto, luego es const,
        luego puede ser llamada por la función del main.cpp que
       es const
    { //regla de oro, todas las funciones miembro de una clase
       que no modifiquen los datos de la clase, marcarlas como
       const. Porque nos va a permitir ejecutar ese método
       desde variables const.
        return cuenta;
    }
    void haz_igual(Contador &otro) const ; //modifica al otro
       objeto, haz igual modifica el objeto que le llama? no,
       entonces es const
};
```

Contador.cpp:

```
#include "Contador.h" // Incluimos la clase
#include <iostream>

void Contador::incrementa() {// Función incrementa que
   pertenece a la clase Contador. Se incluye el ámbito
    cuenta++;
    std::cout << valor() << std::endl;
}

void Contador::haz_igual(Contador &otro) const { //ojo, el
   const admite sobrecarga, se verá en un futuro.
    otro.cuenta = cuenta;
}
```

Main.cpp:

```
#include <iostream>
#include "Contador.h"
using std::cout, std::endl; //cada unidad va a tener que hacer
   el using porque es autocontenida.

void print(const Contador &mc1, const Contador &mc2) { //const!!
    //dentro de print, son métodos const
    cout << "c1:" << mc1.valor() << ";"; // Se puede acceder a
       .valor de mc1 pero no a su cuenta, esta es privada.
    cout << "c2:" << mc2.valor() << "\n"; // valor es const
}
```

```
int main() {
    Contador c1, c2;
    print(c1, c2);
    c1.incrementa();
    print(c1, c2);
    c1.incrementa();
    print(c1, c2);
    c1.haz_igual(c2);
    print(c1, c2);
}
```

8.3. Funciones `const`.

Ideas clave:

- Sólo son métodos.
- Cuando se añade la palabra `const` a un método, se añade tanto en la declaración como en la definición del método.
- Un método que no modifica los datos del objeto debe ser `const`.
- `const` sobrecarga las funciones. Se pueden tener dos funciones de igual nombre y se diferencian por el carácter de sus variables o por si son `const`. En función de si el que ejecuta es `const` o no, definirá una preferencia por la función que se va a utilizar. Ejemplo:

```
int valor () const {
    return contador;
}
int valor() {
    cout<<"No const";
    return contador;
}
```

8.4. El puntero `this`. Palabra clave.

Todos los objetos y todos los métodos de un objeto tienen un parámetro que no vemos que podemos utilizar y es el puntero `this`. Se usa cuando tengo que pasarme a mí mismo o retornarme a mí mismo.

```
class Contador {

    int valor() {
        this; // equivalente a decir '&yo;' 'mi dirección'
        Contador copia (*this);
        (*this).cuenta;
        this->cuenta;
    }
};
```

8.5. Clases y métodos amigos. Palabra clave `friend`.

Es un mecanismo mediante el cual una clase da permiso a clases externas a acceder a su información privada.

¿Quién puede habilitar este acceso? Solo el dueño de los datos puede dar permiso a alguien a acceder a los datos. Ideas clave:

- La amistad no se transfiere. Si una clase A es amiga de B y la clase B es amiga de C, no quiere decir que C sea amiga de A y pueda acceder a la parte privada de A.
- La amistad no se hereda. Si una clase A le da permiso a una clase B y B tiene una clase hija C, entonces C no tiene permiso para acceder a la parte privada de A.
- La amistad no tiene porqué ser simétrica. Si una clase A le da permiso a una clase B, no quiere decir que le demos permiso a A para acceder a la parte privada de B.
- Las declaraciones de amistad se declaran dentro del cuerpo de la clase y no quedan definidas por las zonas de acceso dentro de una clase.

Existen 3 casos de declaración de amistad:

```
class A{
    friend void modifica(A*a); //A
    friend void B::modifica(A*a); //B
    friend class B; //C
private:
    int x = 0;
},
```

Los casos A y C se denominan declaraciones débiles y el B es una declaración estricta.

- En el caso A, la sentencia se puede traducir a "si existe la función modifica, otorgarle permiso".
- En el caso C, la sentencia se puede traducir a "si existe la clase B, se le otorga permiso".
- En cambio, en el caso B, el compilador sí exige conocer la clase B y tiene que ver la declaración del método modifica en la clase B.

```
class B {
    void modifica(A*a){
        a->x=4;
    }
}
```

8.5.1. Ejemplo.

Declaramos una clase matriz y una de vectores. Punto de partida:

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

class Matrix{
    float m[2][2]{1,0,0,1}; /*(1)*/
public:
    void set(float a, float b, float c, float d){
        m[0][0] = a;
        m[0][1] = b;
        m[1][0] = c;
        m[1][1] = d;
    }
};

class Vector{
    float v[2]{};           /*(2)*/
public:
    void set(float a, float b){
        v[0] = a;
        v[1] = b;
    }
    void print(ostream &co) const{
        co << "(" << v[0] << ", " << v[1] << ")";
    }
};

int main(){
    Matrix m1;
    //m1{1,2,2,-1};         /*(3)*/
    Vector v1;
    m1.set(1, 2, 2, -1);
    v1.set(2,3);
    return 0;
}
```

Comentarios a realizar:

- Línea (1). Inicialización de la matriz.
- Línea (2). Inicialización de todos los elementos a cero.
- Línea (3). Esta inicialización no es posible porque `m` es un atributo privado de la clase `matrix` y además la clase no proporciona un constructor personalizado que acepte esos valores como parámetros.

Se desea implementar una función `multiplica` que realice la multiplicación entre una matriz y un vector. Una forma de hacerlo es mediante una función externa que reciba los argumentos `(m1, v1)` y devuelva el resultado de la operación.

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

class Vector;
class Matrix{
    float m[2][2]{1,0,0,1};
public:
    void set(float a, float b, float c, float d){
        m[0][0] = a;
        m[0][1] = b;
        m[1][0] = c;
        m[1][1] = d;
    }
    friend Vector multiplica(const Matrix &m,const Vector &v);
       /*(1)*/
    Vector multiplica(const Vector &v);                 /*(2)*/
};

class Vector{
    float v[2]{};
public:
    void set(float a, float b){
        v[0] = a;
        v[1] = b;
    }
    void print(ostream &co) const{
        co << "(" << v[0] << ", " << v[1] << ")";
    }
    friend Vector multiplica(const Matrix &m, const Vector &v);
       /*(3)*/
    friend Vector Matrix::multiplica(const Vector &v);  /*(4)*/
    friend class Matrix;                                /*(5)*/
};

Vector multiplica(const Matrix &m, const Vector &v){    /*(6)*/
    Vector ret;
    ret.v[0] = m.m[0][0]*v.v[0] + m.m[0][1]*v.v[1];
    ret.v[1] = m.m[1][0]*v.v[0] + m.m[1][1]*v.v[1];
    return ret;
}

Vector Matrix::multiplica(const Vector &v){ //metodo de matrix
    Vector ret;
    ret.v[0] = m[0][0]*v.v[0] + m[0][1]*v.v[1];
    ret.v[1] = m[1][0]*v.v[0] + m[1][1]*v.v[1];
    return ret;
```

```
}

int main(){
    Matrix m1;
    Vector v1;
    m1.set(1, 2, 2, -1);
    v1.set(2,3);
    auto v2 = multiplica (m1,v1);
    auto v3 = m1.multiplica(v1);
    v2.print(cout);
    return 0;
}
```

Comentarios a realizar:

- Línea (1). Hay que decirle a la clase `matrix` que existe la clase `vector`, luego se añade `friend` a la función. El compilador necesita saber que es un tipo de datos.
- Línea (2). Método `multiplica` de la clase `matrix`.
- Línea (3). Opción 1, función externa amiga.
- Línea (4). Opción 2. Se elige una de las dos opciones. Esta opción no se puede usar si no se ve el método en la clase `matrix`.
- Línea (5). Opción 3. Es una versión más sencilla de la opción 1. Pueden estar ambas opciones escritas y no da error.
- Línea (6). En caso de objetos grandes, es recomendable el paso por referencia, con el compromiso de no ser modificado el objeto. Se emplea `const`.

8.6. Concepto de alineación de datos.

Vamos a suponer una clase

```
struct S{
    short a;  //16bits
    int b;    //32bits
    char c,d; //8bits x 2
};
//Si creamos un objeto de tipo S
```

9. Sesión 9. Construyendo y destruyendo objetos.

9.1. Constructor.

Un constructor es un método especial dentro de una clase que se utiliza para inicializar objetos de esa clase. Características:

- Se llaman igual que la clase.
- No pueden ser estáticos ni ser `const` o `virtual`.
- No tienen valor de retorno.
- Suelen ser públicos.
- No se llaman explícitamente, es decir, al crear la clase, creamos el constructor. Se invocan en la propia inicialización del objeto de manera automática.
- Los constructores que carecen del indicador `explicit`[1], además de servir para construir objetos, sirven como medio de conversión (permiten inicializar un objeto de una clase con un valor de otro tipo, y el constructor se encargará de la conversión).
- Se pueden generar tantos constructores como queramos, de 0 a ∞.
- Siempre que se crea un objeto, se ejecuta un constructor.

Tipos:

- Constructor de conversión. Cualquier constructor no explícito con uno o más parámetros.
- Constructor por defecto. El constructor que se puede ejecutar sin argumentos.

```
T::T()
```

- Constructor de copia. Son los constructores que utilizan como primer parámetro un objeto del mismo tipo (&, `const` &) y que el resto de parámetros pueden asumir valores por defecto o que no los hay.

```
T::T(const T&)
```

- Constructor de movimiento. Constructor que toma una referencia rvalue (`T&&`) como primer parámetro. Transfiere (mueve) los recursos de un objeto temporal.

```
T::T(T&&)
```

- Constructor delegante. Constructor que delega su inicialización en otro constructor de la misma clase usando la sintaxis de lista de inicialización.

```
T::T(int x) : T() { /* código adicional */ }
```

- Constructor de oficio. Es el constructor que genera el compilador automáticamente si nosotros no definimos uno.

[1] `explicit` fuerza a que no se hagan conversiones automáticas

Ideas sobre su funcionamiento.

A. Si no hemos definido un constructor, se va a asignar el constructor de oficio por defecto. ¿Este qué hace?

   1. Llama a los constructores por defecto de las clases base.
   2. LLama por orden de declaración a los inicializadores por defecto de los miembros no estáticos de la clase.
   3. A todos los efectos, el constructor por oficio equivale a la codificación de un constructor vacío[2].

      ```
      T::T(){};
      ```

   4. Si el objeto es `static`, entonces llama a la `zero_inicialization` de los datos fundamentales.

B. En cuanto se define un constructor, se pierde el oficio. Si la clase es trivial (todos los atributos son públicos) se pierde la inicialización de agregados.

C. Podemos forzar la creación del constructor de oficio. En la declaración de la clase escribimos:

   ```
   T() = default;
   ```

D. Si por el contrario, queremos evitar la creación automática de un constructor, escribimos:

   ```
   T(const T&) = delete;
   ```

[2]un constructor vacío no toma ningún parámetro y es aquel que inicializa los miembros de datos de la clase según sus tipos

9.1.1. Ejemplo 1.

No hay constructor, luego, se usa el de oficio.

```
class Punto{
    int x,y;
public:
    void set(int _x,int _y){
        x = _x;
        y = _y;
    }
    void print(){
        cout<<x<<","<<y<<endl;
    }
    //[1]
};

Punto exterior;     //(a)
void main(){
    Punto interior; //(b)
    exterior.print();
    interior.print();
    Punto copia(interior);
    copia.print();  //(c)
    Punto dos(3,4); //(d)
}
```

Comentarios a realizar:

- Línea (a). Es una variable global con sus parámetros sin inicializar, luego, es basura.
- Línea (b). Es una variable global con sus parámetros sin inicializar, luego, es basura.
- Línea (c). Imprime los mismos valores basura que interior.
- Línea (d). `ERROR` porque no existe ningún constructor que acepte dos enteros.

9.1.2. Ejemplo 2.

Añadimos un constructor en [1].

```
class Punto{
    int x,y;
public:
    void set(int _x,int _y){
        x = _x;
        y = _y;
    }
    void print(){
        cout<<x<<","<<y<<endl;
    }
    Punto(int _x, int _y);
};

Punto exterior;     //(a)
void main(){
    Punto interior; //(b)
    exterior.print();
    interior.print();
    Punto copia(interior);
    copia.print();  //(c)
    Punto dos(3,4); //(d)
}
```

Si compilamos:

- Línea (a) `ERROR` : Al definir un constructor con parámetros y no definir el constructor por defecto, el compilador no lo genera automáticamente. Por tanto, no se puede instanciar un objeto sin argumentos.
- Línea (b) `ERROR` : Igual que en la línea [a], se intenta usar un constructor por defecto que no existe, lo que provoca un error de compilación.
- Línea (c) `OK` : Esta línea utiliza el constructor de copia. Como no se ha definido uno explícitamente, el compilador lo genera automáticamente. Compila y funciona, aunque puede copiar valores no inicializados.
- Línea (d) `OK` : Se usa el constructor definido por el programador que recibe dos argumentos. Compila correctamente y construye el objeto con los valores proporcionados.

Consecuencias adicionales, al escribir en el main:

```
Punto mipunto;                    //no permitido
Punto puntos[2];                  //no permitido
Punto puntos[2]{ {2,3}, {1,4} }; //correcto
```

9.1.3. Ejemplo 3.

Elimino el constructor de copia de oficio. Añadimos en [1]:

```
class Punto{
    int x,y;
public:
    void set(int _x,int _y){
        x = _x;
        y = _y;
    }
    void print(){
        cout<<x<<","<<y<<endl;
    }
    Punto (const Punto &) = delete;
};
```

```
Punto exterior;     //(a)
void main(){
    Punto interior; //(b)
    exterior.print();
    interior.print();
    Punto
       copia(interior);
    copia.print();  //(c)
    Punto dos(3,4); //(d)
}
```

Si compilamos:

- Línea (a) `ERROR` : Al definir un constructor con parámetros y no definir el constructor por defecto, el compilador no lo genera automáticamente. Por tanto, no se puede instanciar un objeto sin argumentos.
- Línea (b) `ERROR` : Igual que en la línea [a], se intenta usar un constructor por defecto que no existe, lo que provoca un error de compilación.
- Línea (c) `ERROR` : Se ha eliminado el constructor de copia (`Punto(const Punto&)`), por lo que no está permitido copiar objetos de tipo `Punto`. Esta línea provoca un error de compilación.
- Línea (d) `OK` : Se usa el constructor definido por el programador que recibe dos argumentos. Compila correctamente y construye el objeto con los valores proporcionados.

Consecuencias adicionales, al escribir en el main:

```
interior = exterior;    //correcto, se trata de una asignación
   entre objetos ya construidos. Como el operador de asignación
   no ha sido eliminado, el compilador lo genera
   automáticamente.
Punto a = interior;     //no permitido, esta línea implica la
   construcción de un nuevo objeto a partir de otro
   (inicialización por copia), lo cual requiere el constructor
   de copia, que ha sido eliminado explícitamente. Provoca un
   error de compilación.
```

9.1.4. Ejemplo 4.

Se desea generar un constructor de copia por defecto. Para que el código compile correctamente, tenemos tres posibles opciones que pueden incluirse en [1]:

```
class Punto{
    int x,y;
public:
    void set(int _x,int _y){
        x = _x;
        y = _y;
    }
    void print(){
        cout<<x<<","<<y<<endl;
    }
    //[1]
};
```

```
a) Punto() = default;   //
   [a] y [b] válidos
b) Punto() { x = y = 1; }
c) Punto(int _x = 0, int _y
   = 0) { x = _x; y = _y; }
```

```
Punto exterior;     //(a)
void main(){
    Punto interior; //(b)
    exterior.print();
    interior.print();
    Punto copia(interior);
    copia.print();  //(c)
    Punto dos(3,4); //(d)
}
```

9.1.5. Ejemplo 5.

Si en [1] tenemos:

```
Punto (vector<int> &v){
    x = v[0];
    y = v[1];
}
Punto(int k){
    x = k, y = 2*k;
}
```

Entonces puedo decir:

```
vector<int> v{1,2,3,4};
Punto c = v,d;
d = v;
Punto h(2); //h.x=2, h.y=4
Punto l;
l=2;
```

Por otro lado, si se pone en [1]:

```
explicit Punto (int k){
    x = k, y = 2*k;
}
```

Entonces:

```
l=2; //ERROR
```

9.2. Inicialización de atributos.

Método A) Inicialización mediante lista de inicialización en constructor.

La sintaxis general es:

```
T::T(argumentos) : id1(args1), id2(args2), ... {
    // código del constructor
}
```

Ejemplo:

```
class Punto
{
    int x, y;
public:
    Punto(int a, int b)
    {
        x = a;
        y = b;
    }
};
```

```
class Segmento {
    Punto uno;
    Punto dos;
public:
    Segmento(int x1, int y1,
       int x2, int y2) :
       uno(x1, y1), dos(x2, y2){
        // cuerpo vacío
    }
};
```

Método B) Inicialización directa de miembros (C++11).

C++11 introdujo la posibilidad de inicializar los atributos directamente en su declaración, mediante inicializadores por defecto. Si además se provee una lista de inicialización en el constructor, esta última tiene prioridad. Ejemplo:

```
class Punto
{
    int x, y;
public:
    Punto(int a, int b)
    {
        x = a;
        y = b;
    }
};
```

```
class Segmento {
    Punto uno{0,0};  /*(1)*/
    Punto dos{0,0};  /*(2)*/
public:
    Segmento(int x1, int y1,
       int x2, int y2) :
       uno(x1, y1), dos(x2, y2)
       {/*(3)*/
    }
};
```

Comentarios a realizar:

- Línea (1) y (2). Inicialización por defecto.
- Línea (3). Esta inicialización tiene prioridad sobre la inicialización por defecto. Además, se descompone en dos partes:

```
(int x1, int y1, int x2, int y2) //define las variables
uno(x1, y1), dos(x2, y2) //se inicializan los atributos
```

Nota. Mediante inicializaciones por defecto, si no hay constructor por defecto en un atributo o en una referencia miembro o en una constante, es obligatorio poner algún tipo de inicialización.

Otro ejemplo adicional:

```
struct S {
    int n = 7;
    int d{3};
    int e = {3}; /*(1)*/
    Punto uno;
    Punto dos;
    S(int x) : dos(x, x), e{x} { /*(2)*/
        n = x * 2;
    }
};
```

Comentarios a realizar:

- Línea (1). La variable `e` se inicializa con `e{x}` porque la lista de inicialización del constructor tiene prioridad sobre la inicialización directa (línea (2)).
- Línea (2). En esta lista de inicialización se definen los valores para los atributos `dos` y `e`. El orden en la lista no cambia el orden real de inicialización, que sigue el orden de declaración en la clase.

Otro ejemplo adicional:

```
struct F {
    int a;
    int &c;            /*(1)*/
    const int val;
    const int val2 = 2;
    int &b = a;        /*(2)*/
    Punto p{1, 3};
    F() : c(a), val(3) { }
};
```

Comentarios a realizar:

- Línea (1). Tanto las referencias (`int &c`) como las constantes (`const int val`) es obligatorio inicializarlas mediante lista de inicialización en el constructor.
- Línea (2). La variable `b` es una referencia que se inicializa directamente en la declaración. Esto es válido en C++11 y posteriores, y no depende del orden en la estructura.

Recordatorio: La construcción por defecto de un objeto en C++ debe escribirse como:

```
Segmento s1;    // correcto
Segmento s2{};  // correcto (C++11 en adelante)
Segmento s3();  // incorrecto: declara una función que
   devuelve un Segmento
```

9.3. Constructor delegante.

C++11 introdujo la posibilidad de que un constructor delegue su ejecución a otro constructor de la misma clase. Cuando se utiliza un constructor delegante, no se pueden inicializar otros atributos directamente en la lista de inicialización; toda la inicialización se realiza en el constructor al que se delega. Sin embargo, sí es posible ejecutar código adicional dentro del cuerpo del constructor (entre llaves).

Ejemplo básico:

```
class Punto {
    int x, y;
public:
    Punto(int a, int b) : x(a), y(b) {
    }
};

class Segmento {
    const double pi = 3.1415;
    Punto uno{0,0};
    Punto dos{0,0};
public:
    Segmento(int x1, int y1, int x2, int y2) : uno(x1, y1),
       dos(x2, y2) {
    }
};
```

Supongamos ahora que queremos construir un `Segmento` a partir de un punto inicial y un largo horizontal. Podemos usar un constructor delegante:

```
class Segmento {
    /* ... */
public:
    Segmento(int x1, int y1, int x2, int y2) : uno(x1, y1),
       dos(x2, y2) { }

    Segmento(int x, int y, int largo) : Segmento(x, y, x +
       largo, y) {
        // Código adicional si se desea
    }
};
```

Ideas:

- El constructor con tres parámetros delega completamente al constructor con cuatro parámetros.
- El cuerpo del constructor delegante puede contener código adicional, pero no puede incluir más inicializaciones de miembros.
- **Regla de oro del constructor delegante: la única inicialización que puede aparecer, es la del constructor delegante.**

9.4. Constructor de copia.

El constructor de copia es un tipo especial de constructor que define cómo se crea un objeto como copia de otro del mismo tipo. Es fundamental cuando queremos controlar el comportamiento de copiado, especialmente si la clase gestiona recursos dinámicos (memoria, archivos, etc.).

Se puede definir de varias formas, aunque las dos más habituales son:

```
// Opción A (no recomendada)
T::T(T &t) {
    // código del constructor
}
```

```
// Opción B (recomendada)
T::T(const T &t) {
    // código del constructor
}
```

Nota: La opción A (sin `const`) impide copiar objetos `const`, por lo que rara vez se utiliza. Además, no puede coexistir con un constructor del tipo `T::T(T)` (por valor), el cual tampoco se recomienda porque genera copias innecesarias.

Recordatorio: Los atributos de una clase (no estáticos) son únicos para cada objeto. Si dos objetos tienen un atributo `x`, cada uno tendrá su propia copia. Para que un atributo sea compartido entre todos los objetos, debe ser declarado como `static`.

Ideas clave:

- Si no definimos un constructor de copia, el compilador genera uno por defecto. Este llama al constructor copia de la clase base y luego realiza una copia miembro a miembro llamando recursivamente al constructor de copia de cada atributo (no estático) en orden de declaración.
- Si definimos explícitamente un constructor de copia, dejamos de usar el generado automáticamente. Entonces es nuestra responsabilidad copiar correctamente todos los recursos (por ejemplo, memoria dinámica).
- Se ejecuta en situaciones como:
  - Al crear un objeto como copia de otro:

    ```
    T x;        //tres formas equivalentes:
    T a(x);     // copia directa
    T a{x};     // inicialización uniforme (C++11)
    T a = x;    // copia implícita
    ```

  - Al pasar parámetros por valor:

    ```
    void f(T a); // se copia el objeto
    ```

- Al retornar objetos por valor:

```
T f() {
    T temp;
    return temp;
}
```

- Es imprescindible definirlo cuando la clase gestiona recursos externos (como punteros, archivos, etc.), para evitar copias superficiales que provoquen errores o fugas de memoria.
- A partir de C++11 podemos:

```
T(const T&) = default; // Solicita que el compilador lo
   genere automáticamente
T(const T&) = delete;  // Prohíbe la copia
```

- Desde C++17 se introduce la elisión de copias (copy elision). Esto permite al compilador evitar copias innecesarias en ciertas situaciones como el retorno de objetos locales o la construcción de temporales directamente en su destino final. Ejemplo:

```
Punto cuadrado(int x, int y) {
    return Punto(x * x, y * y); // el objeto se construye
       directamente
}
```

Ejemplo:

```
#include <iostream>
using namespace std;
class Punto {
    int x, y;
public:
    Punto(int a, int b) : x(a), y(b) {}

    void mostrar() const {
        cout << "(" << x << "," << y << ")\n";
    }
};

int main() {
    Punto p1(1, 2);
    Punto p2 = p1; // Se llama al constructor de copia por
       defecto
    p2.mostrar();  // Salida: (1,2)
}
```

9.5. Destructor.

Un destructor es una función especial dentro de una clase que se invoca cuando finaliza la vida de un objeto. Su objetivo principal es liberar recursos o realizar tareas de limpieza antes de que el objeto sea destruido. La sintaxis disponible es la siguiente:

```
// Declaración
~T();

// Definición
T::~T() {
    // Código del destructor
}
```

```
class Objeto {
public:
    ~Objeto() {
        cout << "Eto eh el
            fin\n";
    }
};
```

Ideas clave:

- Cada clase tiene, como máximo, un único destructor.
- No acepta argumentos.
- No retorna ningún valor (ni siquiera `void`).
- No puede ser heredado, pero puede ser `virtual` para garantizar el comportamiento correcto en herencia polimórfica.
- Generalmente se declara como `public`, aunque no es obligatorio (por ejemplo, puede ser `protected` o `private` en patrones como el singleton).
- Desde C++11 es posible marcar un destructor como `=default` o `=delete` :
  - `=default` indica que se use el destructor por defecto generado por el compilador.
  - `=delete` impide explícitamente la destrucción de objetos de esa clase.
- El orden de destrucción es inverso al de construcción:
  - Primero se ejecuta el código del destructor de la clase actual.
  - Luego se destruyen sus atributos miembro en el orden inverso al de su construcción.
  - Finalmente, se llama al destructor de la clase base (si existe).
- En muchos casos no es necesario definir un destructor personalizado. Solo se requiere si la clase gestiona recursos manuales (como memoria dinámica, archivos o conexiones).

Ejemplo 1: Liberación de memoria dinámica.

```
class Buffer {
    int* datos;
public:
    Buffer(int n) {
        datos = new int[n];
    }

    ~Buffer() {
        delete[] datos;
        std::cout << "Memoria liberada.\n";
    }
};
```

Ejemplo 2: Destructor virtual y herencia.

```
class Base {
public:
    virtual ~Base() {
        std::cout << "Destructor de Base\n";
    }
};

class Derivada : public Base {
public:
    ~Derivada() {
        std::cout << "Destructor de Derivada\n";
    }
};
```

Nota: Si el destructor de la clase base no es `virtual`, al destruir un objeto a través de un puntero a la clase base, no se ejecutará el destructor de la clase derivada, provocando potenciales fugas de memoria.

Ejemplo 3: Destructor por defecto y eliminado (C++11).

```
class A {
public:
    ~A() = default; // El compilador genera el destructor
};

class B {
public:
    ~B() = delete;  // No se puede destruir objetos de B
};
```

10. Sesión 10. Sobrecarga de operadores.

10.1. Concepto.

La sobrecarga de operadores es una característica que permite implementar cómo se comportan los operadores con tipos de datos personalizados, como clases y estructuras. Esto permite implementar operaciones específicas para los objetos de una clase. Por ejemplo, se puede redefinir el operador `+` para que sume dos objetos de una clase determinada.

Esta sobrecarga se realiza mediante la definición de funciones especiales dentro de la clase (o como funciones externas), que especifican cómo se realizará la operación. La sintaxis disponible es:

-	Función externa	Método (miembro)
Prototipo	`<tipo>operator@(args);`	`<tipo>operator@(args);`
Definición	`<tipo>operator@(args) {...}`	`<tipo>T::operator@(args) {...}`

Ideas clave:

- Los siguientes operadores no se pueden sobrecargar:
  - `.`
  - `::`
  - `.*`
  - `?:`
- Sólo pueden sobrecargarse como miembros de una clase los siguientes operadores:
  - `=`
  - `()`
  - `[]`
  - `new`
  - `->`
  - `delete`
- Al menos uno de los operandos debe ser de un tipo definido por el usuario:
  - `class`
  - `union`
  - `struct`
  - `enum`
- Al sobrecargar ciertos operadores, estos pierden sus propiedades especiales, como la evaluación de cortocircuito o la prioridad estándar:
  - `&&`
  - `,`
  - `||`
  - `@=` [3]
- No es posible definir nuevos operadores (sólo se pueden sobrecargar los ya existentes).

[3]Este carácter representa las distintas combinaciones como `+=`, `-=`, `*=`, etc., según se describe en el apartado 3.5.

- La sobrecarga de los operadores `<<` y `>>`, cuando se usan para entrada/salida, debe hacerse como función externa, usando referencias y recibiendo como parámetros el flujo (`ostream` o `istream`) y una referencia constante al objeto.

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

class T {
private:
    int valor;
public:
    T(int v) : valor(v) {}

    // Amiga para que pueda acceder a miembros privados
    friend ostream& operator<<(ostream& os, const T& obj);
};

// Implementación externa del operador <<
ostream& operator<<(ostream& os, const T& obj) {
    os << "T(valor=" << obj.valor << ")";
    return os;
}

int main() {
    T t(42);
    cout << t << endl;  // Salida: T(valor=42)
    return 0;
}
```

- El uso de referencias en la sobrecarga de operadores es especialmente útil con el operador `=` y sus variantes (como `+=`). Cuando el método de sobrecarga devuelve el objeto por referencia, no se devuelve una copia, sino una referencia al objeto que llamó al operador. Esto permite encadenar operaciones y evita la creación innecesaria de copias. En estos casos, es habitual usar el puntero `this` para devolver el objeto actual. Ejemplo:

```
class MiClase {
public:
    int valor{};
    MiClase& operator +=(int incremento) {
        valor += incremento;
        return *this;
    }
};
```

Así, en el `main`:

```
int main() {
    MiClase A;
    A += 5; // Devuelve el objeto A con valor 5
}
```

Operador	¿Como miembro?	¿Como función externa?
`+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `<`, `>`	Sí	Sí
`=`, `()` , `[]`, `->`, `new`, `delete`	Sí	No
`<<`, `>>` (streams)	No	Sí
`++`, `--`	Sí	Sí
`int + Clase`	No	Sí
Operadores lógicos (`&&`, `||`), coma `,`	Sí	Sí
No sobrecargables	`.`, `.*`, `::`, `?:`	

Cuadro 1: Resumen de sobrecarga de operadores

10.2. Funciones miembro vs funciones `friend`.

• **Funciones miembro:** las funciones miembro toman un único parámetro (el operando que aparece a la derecha del operador), mientras que el operando a la izquierda es el objeto desde el cual se invoca la función y que se precisa normalmente modificar/acceder a él.

• **Funciones `friend`:** se utilizan comúnmente cuando el primer operando no es un objeto de la clase. Estas funciones no pertenecen a la clase, pero se les otorga acceso a sus miembros privados mediante la palabra clave `friend`.

Ejemplo 1: Sobrecarga del operador + como función miembro.

```
class Punto {
private:
    int x, y;
public:
    Punto(int x = 0, int y = 0) : x(x), y(y) {}

    // Sobrecarga como función miembro
    Punto operator+(const Punto& otro) const {
        return Punto(x + otro.x, y + otro.y);
    }
};
```

De manera alternativa, la definición fuera de la clase usando `Punto::`:

```
class Punto { /*código*/
    Punto operator+(const Punto& otro) const;
};
Punto Punto::operator+(const Punto& otro) const {
    return Punto(x + otro.x, y + otro.y);
}
```

Ejemplo 2: Sobrecarga del operador + como función `friend`.

```
class Punto {
private:
    int x, y;
public:
    Punto(int x = 0, int y = 0) : x(x), y(y) {}
    // Declaración de la función friend
    friend Punto operator+(const Punto& a, const Punto& b);
};

// Definición externa de la función friend
Punto operator+(const Punto& a, const Punto& b) {
    return Punto(a.x + b.x, a.y + b.y);
}
```

Ejemplo 3: Sobrecarga del operador == como función miembro.

```
class Punto {
private:
    int x, y;

public:
    Punto(int x = 0, int y = 0) : x(x), y(y) {}

    // Sobrecarga como función miembro
    bool operator==(const Punto& otro) const {
        return (x == otro.x && y == otro.y);
    }
};
```

Ejemplo 4: Sobrecarga del operador == como función `friend`.

```
class Punto {
private:
    int x, y;

public:
    Punto(int x = 0, int y = 0) : x(x), y(y) {}

    // Declaración de la función friend
    friend bool operator==(const Punto& a, const Punto& b);
};

// Definición externa de la función friend
bool operator==(const Punto& a, const Punto& b) {
    return (a.x == b.x && a.y == b.y);
}
```

Ejemplo 5: Sobrecarga del operador += como función miembro.

```
class Racional {
private:
    int num, den;

public:
    Racional(int num = 0, int den = 1) : num(num), den(den) {}

    // Sobrecarga como función miembro
    Racional& operator+=(const Racional& otro);
};

// Definición fuera de la clase: uso explícito de Racional::
Racional& Racional::operator+=(const Racional& otro) {
    num = num * otro.den + otro.num * den;
    den = den * otro.den;
    // Aquí podría llamarse a una función normaliza() si
       existiera
    return *this;
}
```

Ejemplo 6: Sobrecarga del operador += como función `friend`.

```
class Racional {
private:
    int num, den;

public:
    Racional(int num = 0, int den = 1) : num(num), den(den) {}

    // Declaración de la función friend
    friend Racional& operator+=(Racional& a, const Racional& b);
};

// Definición externa de la función friend
Racional& operator+=(Racional& a, const Racional& b) {
    a.num = a.num * b.den + b.num * a.den;
    a.den = a.den * b.den;
    // Aquí también podría llamarse a una función a.normaliza()
       si existiera
    return a;
}
```

10.2.1. Ejemplo. Copia de `std::vector`.

`vector.h`

```
#pragma once
class Vector {
    using TIPO = int; // (1)
    int capacidad = 2;
    int num_elem = 0;
    TIPO *data{new TIPO[capacidad]}; // (2)
public:
    Vector(int cap, TIPO init_val = {}); // (3)
    Vector() = default; // (4)
    ~Vector(){ delete[] data; } // (5)
    int size() const { return num_elem; } // (6)
    void push_back(const TIPO &);
    TIPO& at(int i); // (7)
    auto begin() const { return data; } // (8)
    auto end() const { return data + num_elem; } // (9)
};
```

Comentarios a realizar:

- Línea (1). Se utiliza un alias de tipo (`TIPO`) para representar los elementos del vector. Actualmente es un `int`, pero si se cambia esta línea (por ejemplo, a `double`), todos los usos posteriores se adaptan automáticamente. Esto mejora la mantenibilidad del código.
- Línea (2). Se inicializa el puntero `data` usando una lista de inicialización directa con `new TIPO[capacidad]`, lo cual reserva un array dinámico de enteros (o del tipo definido en `TIPO`). Esta reserva de memoria requiere una gestión manual, razón por la cual más adelante se define un destructor.
- Línea (3). Se declara un constructor que permite especificar la capacidad del vector y **opcionalmente un valor inicial para sus elementos**. El parámetro por defecto `{}` inicializa el valor con el valor por defecto del tipo (cero en este caso).
- Línea (4). Se usa la sintaxis `= default` para indicar que el constructor por defecto se utilizará tal como lo generaría el compilador. Esto es útil cuando hay otros constructores definidos pero aún se quiere permitir la construcción sin parámetros.
- Línea (5). Es imprescindible definir un destructor porque `data` apunta a memoria dinámica. Aquí se libera esa memoria con `delete[] data`, evitando fugas de memoria.
- Línea (6). El método `size()` devuelve el número de elementos actualmente almacenados en el vector. Se marca como `const` porque no modifica el estado del objeto.

- Línea (7). El método `at(int i)` devuelve una referencia al elemento en la posición `i`. Usar una referencia permite modificar directamente el valor almacenado. También permite usar este método como lvalue (por ejemplo, `vec.at(2) = 5;`).
- Línea (8). Se define el método `begin()` que devuelve un puntero al primer elemento (`data`). Esto permite que el objeto `Vector` pueda usarse en bucles tipo `for-each` o con algoritmos de la STL.
- Línea (9). El método `end()` devuelve un puntero al elemento justo después del último (`data + num_elem`). Esto también es necesario para compatibilidad con la STL y bucles basados en rango. basados en rango.

`vector.cpp`

```
#include "vector.h"
Vector::Vector(int cap, TIPO init_val):
    capacidad(cap<2? 2:cap),
    num_elem{capacidad}, //4
    data{new TIPO[capacidad]}{
    for(int i = 0; i < num_elem; i++)
            data[i] = init_val;
}
```

Se modifica la cuarta línea para considerar el caso de que la capacidad sea negativa.

```
#include "vector.h"
Vector::Vector(int cap, TIPO init_val): /*(1)*/
    capacidad(cap<2? 2:cap),
    num_elem{cap<0? 0:cap},
    data{new TIPO[capacidad]}{ /*(2)*/
    for(int i = 0; i < num_elem; i++) /*(3)*/
            data[i] = init_val;
}
```

Comentarios a realizar:

- En la línea (1), se definen los inicializadores del constructor.
- La línea (2), realmente no es necesaria porque la inicialización de `data` se hace después de la inicialización de `capacidad` y en la declaración (en `vector.h`), ya la habíamos definido en la línea (3) como:

  ```
  TIPO init_val = {}
  ```

- En la línea (3), una alternativa a este bucle, es haber escrito:

  ```
  for(auto &v:*this)
  v = init_val;
  ```

Ahora implementamos una funcionalidad `push_back`

```
void Vector::push_back (const TIPO &valor){
    if(num_elem == capacidad){
        auto aux = new TIPO[capacidad *= 2];
        for (int i = 0; i < num_elem; i++)
            aux[i] = data[i];
        delete[] data;
        data = aux;
    }
    data [num_elem++] = valor;
}
```

Este método comprueba primero que el vector esté lleno, para después crear otro vector auxiliar que se utilizará para introducir el nuevo valor además de los anteriores. En el bucle `if`, se realiza un `delete` para eliminar la memoria dinámica de `data`.

Implementación de funcionalidad `at`

```
TIPO& Vector::at(int i){
    static TIPO nulo{};
    return(( i >= 0)&&(i < num_elem))?data[i]:nulo;
}
```

Por tanto, el archivo `vector.cpp` queda de la siguiente manera:

```
#include "vector.h"
Vector::Vector(int cap, TIPO init_val):
    capacidad(cap<2? 2:cap),
    num_elem{cap<0? 0:cap},
    data{new TIPO[capacidad]}{
    for(int i = 0; i < num_elem; i++)
        data[i] = init_val;
}

void Vector::push_back (const TIPO &valor){
    if(num_elem == capacidad){
        auto aux = new TIPO[capacidad *= 2];
        for (int i = 0; i < num_elem; i++)
            aux[i] = data[i];
        delete[] data;
        data = aux;
    }
    data [num_elem++] = valor;
}

TIPO& Vector::at(int i){
    static TIPO nulo{};
    return(( i >= 0)&&(i < num_elem))?data[i]:nulo;
}
```

Entonces, teniendo este `main.cpp`

```
#include <iostream>
#include "vector.h"
using namespace std;

int main(){
    Vector V1, V2(5), V3(5,7); // (1)
    for(int i = 1; i < 1000; i *= 2) // (2)
        V1.push_back(i);
    for(auto p : V1) // (3)
        cout << p << ","; // (4)
    cout << endl;
    for(auto p : V2)
        cout << p << ","; // (5)
    cout << endl;
    for(auto p : V3)
        cout << p << ","; // (6)
    cout << endl;
    Vector copia(V2); // (7)
    copia.at(2) = 5;  // (8)
    for(auto p : copia)
        cout << p << ","; // (9)
    cout << endl;
    for(auto p : V2)
        cout << p << ","; // (10)
    cout << endl;
    return 0;
}
```

Comentarios a realizar:

- Línea (1). Se crean tres objetos de tipo `Vector`: `V1` con el constructor por defecto, `V2` con capacidad 5 e inicialización por defecto (0), y `V3` con capacidad 5 y valor inicial 7.
- Línea (2). Bucle `for` que multiplica por 2 en cada iteración. Se insertan los valores `1, 2, 4, ..., 512` en `V1` mediante `push_back`.
- Línea (3). Se usa un bucle basado en rango para recorrer los elementos de `V1`. Se corrigió el identificador a mayúscula.
- Línea (4). Se imprime el contenido de `V1`, esperado como

  ```
  1,2,4,8,16,32,64,128,256,512
  ```

- Línea (5). Se imprime el contenido de `V2`. Al haber sido creado con valor inicial por defecto, los cinco elementos son `0`.

  ```
  0,0,0,0,0
  ```

- Línea (6). Se imprime el contenido de `V3`, que contiene cinco veces el valor `7`.

```
7,7,7,7,7
```

- Línea (7). Se crea un nuevo vector `copia` como copia de `V2`. Se espera que esto realice una copia profunda del array dinámico.
- Línea (8). Se modifica el tercer elemento de `copia` usando `at(2)` para asignarle el valor `5`.
- Línea (9). Se imprime `copia`. Debería mostrar `0,0,5,0,0`, reflejando el cambio realizado solo en esta copia.
- Línea (10). Se imprime `V2` nuevamente para verificar que no fue alterado por la modificación de `copia`.

```
0,0,5,0,0
```

Finalmente, se completa el código con el constructor de copia y la sobrecarga del operador `=` :

```
#pragma once
class Vector {
    using TIPO = int;
    int capacidad = 2;
    int num_elem = 0;
    TIPO *data{new TIPO[capacidad]};
public:
    Vector(int cap, TIPO init_val = {});
    Vector() = default; // (4)
    ~Vector(){ delete[] data; }
    int size() const { return num_elem; }
    void push_back(const TIPO &);
    TIPO& at(int i);
    auto begin() const { return data; }
    auto end() const { return data + num_elem; }
    Vector(const Vector&); //(1)
    Vector &operator = (const Vector &); //(2)
};
```

Comentarios a realizar:

- Línea (1). Se declara un constructor de copia empleado dentro de la función `push_back`. Utiliza la inicialización del primer constructor con un solo argumento, la capacidad del vector `v` que se ha pasado al constructor de copia. La definición es:

```
Vector::Vector(const Vector &v):Vector(v.capacidad){
    for(int i = 0; i<num_elem; i++){
        data[i] = v.data[i];
    }
}
```

- Línea (2). Prototipo de sobrecarga del operador asignación `=`, en el que se especifica que el valor de vuelta es una referencia a un objeto `Vector`. Esta referencia permite modificar el elemento. Su definición es:

```
Vector& Vector::operator=(const Vector& v) {
    if (capacidad < v.num_elem) {
        delete[] data;
        capacidad = v.capacidad;
        data = new TIPO[capacidad];
    }
    num_elem = v.num_elem;
    for (int i = 0; i < num_elem; i++)
        data[i] = v.data[i];
    return *this;
}
```

El `main.cpp` tiene la forma final:

```
#include <iostream>
#include "vector.h"
using namespace std;

int main(){
    Vector V1, V2(5), V3(5,7);
    for(int i = 1; i < 1000; i *= 2)
        V1.push_back(i);
    for(auto p : V1)
        cout << p << ",";
    cout << endl;
    for(auto p : V2)
        cout << p << ",";
    cout << endl;
    for(auto p : V3)
        cout << p << ",";
    cout << endl;
    Vector copia(V2);
    copia.at(2) = 5;
    for(auto p : copia)
        cout << p << ",";
    cout << endl;
    for(auto p : V2)
        cout << p << ",";
    cout << endl;
    copia = V3; //equivalente a copia.operator= (V3)
    return 0;
}
```

10.2.2. Ejemplo. Copia de `std::vector` II.

Considerando que tenemos la clase `Vector` anterior, sobrecargamos el operador "`{}`".

```
class Vector{
    //resto del codigo
    TIPO& operator[](int i){
        return data[i];
    }
    TIPO& operator[](const Vector &);
};
```

Siendo la definición:

```
Vector::TIPO Vector::operator[](const Vector &v){
    TIPO suma{};
    for(auto p:v) suma+=data[p];
    return suma;
}
```

Implementación en el `main.cpp`

```
#include <iostream>
#include "vector.h"
using namespace std;

int main(){
    Vector V1, V2(5), V3(5,7);
    for(int i = 1; i < 1000; i *= 2) V1.push_back(i);
    for(auto p : V1) cout << p << ",";
    cout << endl;
    for(auto p : V2) cout << p << ",";
    cout << endl;
    for(auto p : V3) cout << p << ",";
    cout << endl;
    Vector copia(V2);
    copia.at(2) = 5;
    for(auto p : copia)
        cout << p << ",";
    cout << endl;
    for(auto p : V2)
        cout << p << ",";
    cout << endl;
    copia = V3; //continuación:
    Vector vd(3);
    vd[0] = 1;
    vd[1] = 3;
    vd[2] = 4;
    cout<<v1[vd]<<endl;
    cout<<0b11010<<endl;
    return 0;
}
```

La siguiente cuestión es, cómo mostrar directamente el vector. Se necesita definir una función `friend` en la clase `Vector` para poder sobrecargar el operador `<<`.

```
class Vector{
    //resto del codigo
    TIPO& operator[](int i){
        return data[i];
    }//añadimos función friend
    friend ostream&operator<<(ostream &co, const Vector &v);
};
```

Definición (plantilla):

```
inline ostream&operator<<(ostream &co, const Vector &v){
    co<<"(";
    for(auto p:v)
        co<<p<<",";
    co<<"\b\b)";
    return co;
}
```

Implementación de `main.cpp`

```
int main(){
    Vector V1, V2(5), V3(5,7);
    for(int i = 1; i < 1000; i *= 2) V1.push_back(i);
    for(auto p : V1) cout << p << ",";
    cout << endl;
    for(auto p : V2) cout << p << ",";
    cout << endl;
    for(auto p : V3) cout << p << ",";
    cout << endl;
    Vector copia(V2);
    copia.at(2) = 5;
    for(auto p : copia) cout << p << ",";
    cout << endl;
    for(auto p : V2) cout << p << ",";
    cout << endl;
    copia = V3; //continuación:
    Vector vd(3); //vector de 3 ceros
    vd[0] = 1;
    vd[1] = 3;
    vd[2] = 4;
    cout<<v1[vd]<<endl; //V1 se mete dentro de vd
    cout<<0b11010<<endl;
    //ahora puedo escribir:
    cout<<"v1["<<vd<<"]"="<<V1[vd]"; //sale v1[(1,3,4)] = 27
    return 0;
}
```

10.2.3. Inciso. Initializer_list.

`initializer_list` es una clase de la STL que permite la inicialización uniforme de una lista de elementos mediante llaves `{}`. Esto resulta especialmente útil para proporcionar una sintaxis sencilla y legible al inicializar contenedores personalizados, como nuestra clase `Vector`.

A continuación se muestra cómo integrarla en la clase:

```
#include <initializer_list>

class Vector {
    //etc
public:
    // Constructor que acepta initializer_list
    Vector(std::initializer_list<TIPO> &l):Vector(l.size()){
        num_elem = 0;
        for (auto p : l)
            push_back(p);
    }
};
```

Este constructor permite instanciar un objeto de tipo `Vector` utilizando la sintaxis:

```
Vector vd2{1, 3, 4};
```

Lo cual es equivalente a llamar a `push_back(1)`, `push_back(3)` y `push_back(4)` secuencialmente.

Notas importantes:

- La clase `std::initializer_list<T>` no proporciona métodos para modificar sus elementos, es un contenedor ligero de solo lectura.
- El constructor se inicializa delegando en `Vector(l.size())`, asegurando que el vector tenga capacidad suficiente para todos los elementos.
- La variable `num_elem` se reinicia a cero, ya que el constructor delegante ya asignó la capacidad, pero no insertó elementos.

10.2.4. Sobrecarga de operadores de comparación para la clase `Vector`.

Operador comparación menor que:

```
bool Vector::operator<(const Vector &v) const{
    int i = 0;
    while ((i < num_elem) && (i < v.num_elem)){
        if (data[i] < v.data[i])
            return true; // el vector es más pequeño
        if (data[i] > v.data[i])
            return false;
        i++;
    }
    return num_elem < v.num_elem;
}
```

Operador comparación mayor que:

```
bool operator>(const Vector &v1, const Vector &v2){
    return v2 < v1;//proviene de que a > b es lo mismo que b < a
}
```

Operador mayor o igual que es lo mismo que `!(a <b)`:

```
bool operator>=(const Vector &v1, const Vector &v2){
    return !(v1 < v2);
}
```

Operador menor o igual es lo mismo que `!(a >b)`:

```
bool operator<=(const Vector &v1, const Vector &v2){
    return !(v1 > v2);
}
```

Operador igual:

```
bool Vector::operator==(const Vector &v) const{
    if (num_elem != v.num_elem)
        return false;
    for (int i = 0; i < num_elem; i++)
        if (data[i] != v.data[i])
            return false;
    return true;
}
```

10.2.5. Ejercicio propuesto.

Se tiene el siguiente código inicial:

```
class Tiempo {
    int hora;
    int minuto;
public:
    explicit Tiempo(int h = 0, int m = 0) { /*(1)*/
        if (m < 0)
            m = 0;
        hora = h < 0 ? 0 : (h + m / 60);
        minuto = m % 60;
    }
};
```

Comentarios a realizar:

- Línea (1). El uso de `explicit` evita conversiones implícitas de tipos, impidiendo así que una llamada como `Tiempo t = 10;` compile. Esto reduce el riesgo de errores sutiles al trabajar con constructores con un solo parámetro.

Vamos a realizar la sobrecarga de distintos operadores para la clase:

- Operador suma para dos objetos Tiempo (como método):

```
Tiempo operator+(const Tiempo &t2) const {
    return Tiempo(hora + t2.hora, minuto + t2.minuto);
}
```

- Operador de comparación menor que `<` (como método):

```
bool operator<(const Tiempo &t2) const {
    return (hora < t2.hora) || (hora == t2.hora && minuto <
       t2.minuto);
}
```

- Operador de inserción en flujo de salida. Declaración (dentro de la clase):

```
friend ostream &operator<<(ostream &co, const Tiempo &);
```

  Definición (fuera de la clase):

```
ostream &operator<<(ostream &co, const Tiempo &t) {
    return co << t.hora << ":" << t.minuto;
}
```

- Operador suma para `Tiempo + int` minutos (función externa `inline`):

```
inline Tiempo operator+(const Tiempo &t, int mins) {
    return t + Tiempo(0, mins);
}
```

- Operador suma para `int + Tiempo` (función externa `inline`):

```
inline Tiempo operator+(int mins, const Tiempo &t) {
    return t + Tiempo(0, mins);
}
```

- Operador de igualdad ==. Declaración (en la clase):

```
bool operator==(const Tiempo &) const;
```

  Definición (fuera de la clase):

```
inline bool Tiempo::operator==(const Tiempo &t) const {
    return (hora == t.hora) && (minuto == t.minuto);
}
```

- Operador mayor o igual que >= (función externa):

```
inline bool operator>=(const Tiempo &t1, const Tiempo &t2) {
    return !(t1 < t2);
}
```

En el caso del operador pre-incremento `++t2`. Dos opciones:

Definición como miembro.

```
Tiempo& operator++()
{
    *this = *this + Tiempo(0,
       1);  // Añade 1 minuto
    return *this;
}
```

Definición como función externa.

```
Tiempo& operator++(Tiempo &t)
{
    t = t + 1;  // Utiliza el
       operador+ ya definido
    return t;
}
```

Por otro lado, en el caso del operador post-incremento `t2++`. Dos opciones:

Definición como miembro.

```
Tiempo operator++(int)
{
    Tiempo ret(*this);  //
       Guarda copia del estado
       actual
    ++*this;            //
       Aplica preincremento
    return ret;         //
       Retorna el valor anterior
}
```

Definición como función externa.

```
Tiempo operator++(Tiempo &t,
   int)
{
    Tiempo ret(t);  // Guarda
       copia del estado actual
    ++t;            // Aplica
       preincremento
    return ret;     // Retorna
       el valor anterior
}
```

Para incluir el operador de extracción de flujo de entrada, se implementa una función externa

```
#include <string>
using std::string;

istream &operator>>(istream &ci, Tiempo &t){
    string a;
    ci >> a;  // Lee la cadena del flujo de entrada
    int index = a.find_first_of(":");
    if(index != string::npos)
    {
        t.hora = stoi(a.substring(0, index)); /*(1)*/
        t.minuto = stoi(a.substring(index + 1)); /*(2)*/
    }
    return ci;
}
```

Comentarios a realizar:

- Línea (1), extrae las horas y la línea (2) extrae minutos.
- Línea (1) y (2), `stoi` es una función específica de la librería `string` que convierte una cadena de caracteres a enteros.

Finalmente, para incluir un convertidor de tipo, se debe incluir dentro de la clase:

```
class Tiempo{
    int hora;
    int minuto;
public:
    explicit Tiempo(int h = 0, int m = 0){/*(1)*/
        if(m < 0)
            m = 0;
        hora = h < 0 ? 0 : (h + m / 60);
        minuto = m%60;
    }

    operator int()
    {
        return hora*60 + minuto;
    }
};
```

10.3. Functor.

`Functor` es un objeto que sobrecarga el operador `()` para que pueda ser utilizado como si fuera una función. Ejemplo:

```
#include <iostream>
using namespace std;
struct MiFunctor {
    void operator()() const {
        cout << "Ejecutando el functor." << endl;
    }
};

int main() {
    MiFunctor f;
    f(); // Llama al operador ()
    return 0;
}
```

También se pueden devolver estructuras desde el operador `()`:

```
#include <iostream>
using namespace std;
struct Reloj {
    int hora;
    int minuto;
    auto operator()() const {
        struct {
            int a, b;
        } tiempo{hora, minuto};
        return tiempo;
    }
};

int main() {
    Reloj r{14, 45}; // 14:45
    auto resultado = r();
    cout << "Hora: " << resultado.a << endl;
    cout << "Minuto: " << resultado.b << endl;
    return 0;
}
```

10.4. Miembros static.

Dentro de la definición de una clase, se puede anteponer la palabra clave `static` a cualquier miembro (atributo o método). Esto indica que dicho miembro no pertenece a las instancias de la clase, sino a la clase en sí. Es decir, los miembros estáticos son compartidos por todos los objetos de esa clase. Ideas clave:

- La palabra `static` se coloca en la declaración del miembro, no en su definición (cuando esta se realiza fuera de la clase).
- Antes de C++17, los atributos estáticos debían definirse en un archivo `.cpp` asociado a la clase. A partir de C++17, pueden definirse directamente en la declaración `.h` con la palabra clave `inline`, por ejemplo:

```
class X1 {
    inline static int x = 3;
};
```

- Un miembro estático puede tener cualquier especificador de acceso: `public`, `protected` o `private`.
- El acceso a un miembro estático `m` de una clase `X` se puede realizar de varias maneras:
  - A través de un objeto `E` de la clase `X`:

`E.m`	`pE->m`

  - Mediante el nombre de la clase:

```
X::m
```

  - Dentro de un método de la propia clase `X`, puede accederse directamente como `m`, sin necesidad de prefijo.
- Los métodos estáticos también son independientes de los objetos. No pueden acceder a miembros no estáticos directamente y no disponen del puntero `this`, ya que no hay una instancia concreta asociada a su ejecución.

10.4.1. Ejemplo 1.

Se tiene el siguiente código.

```
#include<iostream>
#include<vector>
using std::cout, std::endl, std::vector;

struct Punto{
    friend class Factoria; /*(1)*/
    int x,y;
private: /*(2)*/
    Punto(int _x, int _y) : x(_x), y(_y){}
    ~Punto(){}
};

class Factoria{
    static vector<Punto*> v_puntos; /*(3)*/
public:
    static int get_num_puntos(){return v_puntos.size();}
    static Punto *create_punto(int x, int y){
        v_puntos.push_back(new Punto(x,y)); /*(4)*/
        return v_puntos.back();
    }
    static void delete_punto(Punto *p){
        for (auto i=begin(v_puntos); i != end(v_puntos); i++)
            if(p==*i){
                v_puntos.erase(i);
                break;
            }
        delete p;
    }
    static void delete_all(){
        for(auto p:v_puntos) delete p;
        v_puntos.clear();
    }
};

int main(){
    //Punto p1(2,3); /*(5)*/
    auto p2 = Factoria::create_punto(2,3);
    auto p3 = Factoria::create_punto(3,3);
    auto p4 = Factoria::create_punto(4,3);
    cout<<Factoria::get_num_puntos()<<endl;
    Factoria::delete_punto(p2); /*(6)*/
    cout<<Factoria::get_num_puntos()<<endl;
    Factoria::delete_all();
    cout<<Factoria::get_num_puntos();
    //auto p5 = new Punto(p2); /*(7)*/
}
```

Comentarios a realizar:

- Línea (1). `Factoria` puede acceder a la parte privada de la clase. Esto permite que la factoría cree y destruya objetos de tipo `Punto`, aunque su constructor y destructor sean privados.
- Línea (2). Llama la atención que el constructor y el destructor sean privados. De este modo se impide crear o destruir objetos `Punto` directamente fuera de la clase o sin pasar por la `Factoria`. Se fuerza así el uso del patrón factoría.
- Línea (3). Con las variables estáticas no se necesita definir un objeto previamente. Por ejemplo, se podría escribir:

```
Factoria mi_factoria;
//luego escribir:
mi_factoria.metodo;
```

- Línea (4). Crea los puntos. Se usa `new` porque el constructor es privado y solo puede llamarse desde la clase `Factoria`. Cada puntero se guarda en el vector `v_puntos` para tener un registro de todos los objetos creados y poder gestionarlos (borrarlos, contarlos, etc.). La función devuelve un puntero al último elemento creado.
- Línea (5). No se puede invocar directamente al constructor de `Punto`.
- Línea (6). Se observa que se no se puede hacer `delete p2` directamente, ya que la clase `Factoria` mantiene el control de los objetos creados. El destructor es privado y no se ejecutará a menos que sea llamado por `Factoria`. Por eso se debe usar `delete_punto`.
- Línea (7). Aunque el constructor sea privado, el constructor de copia no ha sido eliminado explícitamente, por lo que técnicamente aún podría usarse (lo cual es peligroso si se hace un `new Punto(p2)`). Sería buena práctica declararlo como `delete` para evitar errores.

Si realizamos la siguiente modificación:

```
class Factoria{
    inline static vector<Punto*> v_puntos; /*(1)*/
    static int kk; /*(2)*/
public:
    static int get_num_puntos(){return v_puntos.size();}
    static Punto *create_punto(int x, int y){
        v_puntos.push_back(new Punto(x,y));
        return v_puntos.back();
    }
    static void delete_punto(Punto *p){
        for (auto i=begin(v_puntos); i != end(v_puntos); i++)
        if(p==*i){
            v_puntos.erase(i);
            break;
```

```
        }
        delete p;
    }
    static void delete_all(){
        for(auto p:v_puntos) delete p;
        v_puntos.clear();
    }
};

vector<Punto*> Factoria::v_puntos; /*(1)*/
int Factoria::kk = 0; /*(2)*/
```

Comentarios a realizar:

- Línea (1). Un vector que contiene elementos tipo puntero a punto, a partir de C++17 el uso de `inline` hace que la variable quede registrada como `static` sin tener que volver a llamarla fuera de la clase. La definición del método es:

  ```
  vector<Punto*> Factoria::v_puntos; //definición
  ```

- Línea (2). Se añade una variable `static` llamada `kk` para ejemplificar un caso contrario al comentario anterior. En este caso, sí es necesario añadir una definición obligatoriamente. Por eso se añade:

  ```
  int Factoria::kk = 0;
  ```

10.4.2. Ejemplo 2. Cadenas de caracteres.

```
#include <string>
#include <iostream>
#include <cstring>
using std::string, std::cout, std::endl;

int main() {
    string hw = "hola";
    string s = hw;
    hw += " Mundo";                 /*(1)*/
    auto s2 = s + "Mundo";          /*(2)*/
    cout << hw[4] << endl;          /*(3)*/
    cout << strlen(s.c_str()) << endl;  /*(4)*/
    string q{"En una dacha"};
    q.erase(5, 1);                      /*(5)*/
    q.insert(6, "lugar de la ");        /*(6)*/
    q.append("...");                    /*(7)*/
    auto pos = q.find("dacha");         /*(8)*/
    if (pos != string::npos) {
        q.replace(pos, 2, "Man");       /*(9)*/
    }
    cout << q << endl;
```

```
    string s1("Maria"), s3("Pedro");
    if (s1 == s3)                        /*(10)*/
        cout << "Son iguales" << endl;
    if (s1 < s3)                         /*(11)*/
        cout << s1 << " es anterior a " << s3 << endl;
    return 0;
}
```

- Línea (1). Se modifica la cadena `hw` usando el operador `+=`, añadiendo el texto `Mundo`. La variable `hw` pasa de ser `hola` a `hola Mundo`.
- Línea (2). Se crea una nueva cadena `s2` a partir de la concatenación de `s` y `Mundo`. La cadena original `s` no se modifica. Es un ejemplo del operador `+` aplicado a `std::string`.
- Línea (3). Se imprime el carácter en la posición 4 de `hw`, que es el espacio entre `hola` y `Mundo`.
- Línea (4). Se usa `strlen(s.c_str())` para obtener la longitud de la cadena estilo `C` subyacente de `s`. Equivale a `s.size()` pero usa una función de la librería `C` (`cstring`).
- Línea (5). Se elimina 1 carácter desde la posición 5 de la cadena `q`, eliminando el espacio entre `una` y `dacha` → resultado: `En un dacha`.
- Línea (6). Se inserta la cadena `lugar de la ` en la posición 6 de `q`, obteniendo la frase `En un lugar de la dacha`.
- Línea (7). Se añaden puntos suspensivos al final de la cadena mediante `append`. Resultado: `En un lugar de la dacha...`.
- Línea (8). Se busca la subcadena `dacha` en `q` usando `find`. Si se encuentra, se devuelve la posición donde comienza.
- Línea (9). Si `dacha` se encuentra, se reemplazan sus dos primeras letras (`da`) por `Man`, dando como resultado: `En un lugar de la Mancha...` (referencia al Quijote).
- Línea (10). Se compara si las cadenas `s1` y `s3` son iguales, usando el operador `==`.
- Línea (11). Se compara lexicográficamente si `s1` es menor que `s3`. Si lo es, se imprime que `s1` es anterior (en orden alfabético) a `s3`.

11. Sesión 11. La Herencia.

11.1. Concepto.

La herencia es una de las características fundamentales de los sistemas orientados a objetos. Es un mecanismo que permite definir una clase derivada/hija a partir de una o más clases ya existentes, llamadas clase(s) base(s) o clases padre. Esta modificación, normalmente es una agregación, es decir, define una clase agregándole datos y métodos a otra clase base.

Los niveles de acceso disponibles en la herencia:

- `public` : un método público es accesible a los miembros de la clase y objetos de otras clases.
- `private` : un método privado solamente es accesible a los métodos de la clase y las clases definidas como `friend`.
- `protected` : un método protegido es accesible para los miembros, los amigos y los métodos de clases hijas.

Luego, la herencia depende de los niveles de acceso definidos. Conceptualmente:

- La herencia pública (la más habitual) refleja la relación "is a". Todo lo que se hereda, se hereda con el mismo nivel de acceso que tenía. Lo que era público, seguirá siendo público. Lo que era privado, pues seguirá siendo privado.
- La herencia protegida deja de ser una relación "is a" y pasa a ser una relación "está compuesto por o está implementado por". Luego, lo que era público y protected pasan a ser protected. Lo que era privado, sigue siendo privado.
- La herencia privada, lo que era publico y protegido pasa a ser privado. Si algo antes era privado en la clase base, no es observable por la clase hija. Es decir, la privacidad es sagrada y no se puede acceder a través de la herencia.

La sintaxis que tenemos disponible para la declaración es:

```
class<id_clase_derivada>:public<id_clase_base>{
    //código
};
```

Comentarios a realizar:

- En la siguiente línea, se puede escribir o `class` o `struct`.

  ```
  struct<id_clase_derivada>:public<id_clase_base>
  ```

- La definición del nivel de acceso es opcional y existen tres opciones:

  ```
  public   private    protected
  ```

- En caso de definir la declaración como `class`, entonces como se ha estudiado previamente, los métodos/atributos definidos son privados por defecto y no se pueden usar en clases hijas. Ejemplo:

```
class B: public A{
    //atributos privados
};
```

- En caso de definir la declaración como `struct`, entonces como se ha estudiado previamente, los métodos definidos con públicos por defecto.

En base a los comentarios anteriores, estas dos definiciones son equivalentes:

```
class B:A{

};
```

```
class B: private A{

};
```

Otras definiciones equivalentes:

```
struct B:A{

};
```

```
struct B: public A{

};
```

Cosas que no se heredan:

- No se heredan los constructores de la clase base. Sí se pueden usar pero no podemos construir objetos de clase derivada con la interfaz de los constructores de la clase base. Para construir objetos de la clase derivada no puedo usar la sintaxis de la clase base. Ejemplo ilustrativo:

```
struct A{
    int b;
    A(int x):b(x){}
};

struct B:A{
    int c;
    B(int x, int y):A(x), c(y){}
};

B x(12,15); //correcto
A x(13); //correcto
B x(12); //incorrecto, no se hereda la interfaz de A
```

Sin embargo, en C++14 se pueden resucitar los constructores y se hace de la siguiente manera:

```
struct B:A{
    int c;
    B(int x, int y):A(x), c(y){}
    using A::A; //se añade
};
```

- No se heredan ni las funciones friend ni las relaciones `friend`.
- Tampoco se heredan ni las funciones y ni los atributos estáticos, pero sí se pueden usar.
- No se heredan las operaciones de asignación/copia sobrecargadas. Si no defines la operación de asignación, se te da la de asignación por defecto. Este, llama al operador de la clase base y luego llama a los operadores dentro de la clase.

Ideas importantes:

- Lo que es privado en la clase no es accesible de ninguna forma desde las clases derivadas. Se respeta la privacidad.
- El tipo de herencia cambia el carácter de lo heredado para los herederos y para los externos. Pero el que hereda mantiene los derechos.
- Modificador `final`. A partir de C++11 se incluye la palabra clave `final` que es clave solamente en el ámbito de la declaración de clases, fuera no. Sirve para indicar el punto a partir del cual no se puede heredar de una clase.

```
class A final{
    //codigo
}

class B{
    //codigo
}

class C final: B {
    //codigo
}

class D:A {    //error
    //codigo
}

class E:C{      //error
    //codigo
}
```

11.1.1. Ejemplo 1.

Se tienen los siguientes códigos. Indicar qué sentencias dan error.

```
class A{
    int a;
protected:
    int b;
    public:
    int c;
};
```

Luego, se tiene una clase B que hereda públicamente de A.

```
class B: public A{
    int d;
public:
    B(){
        a=1;
        b=2;
        c=3;
        d=4;
    }
};
```

Luego, se tiene una clase C que hereda públicamente de B.

```
class C:public B{
    int e;
public:
    C(){
        a = 1;
        b = 2;
        c = 3;
        d = 4;
        e = 5;
    }
};
```

Se tiene un main:

```
void main(){
    B b;
    b.a = 0;
    b.b = 1;
    b.c = 2;
    b.d = 3;
}
```

Solución:

```
class B:public A{
    int d;
public:
    B(){
        a=1; //incorrecto
        b=2; //ok
        c=3; //ok
        d=4; //ok
    }
};

class C:public B{
    int e;
public:
    C(){
        a = 1; //ok
        b = 2; //ok, mantiene su caracter protected
        c = 3; //ok
        d = 4; //no, es privado de B
        e = 5; //ok
    }
};

void main(){
    B b;
    b.a = 0; //incorrecto
    b.b = 1; //no , b es protected en A
    b.c = 2; //ok
    b.d = 3; //no
}
```

11.1.2. Ejemplo 2.

Se tienen los siguientes códigos. Indicar qué sentencias dan error.

```
class A{
    int a;
protected:
    int b;
    public:
    int c;
};

class B:private A{ //en este caso, la herencia es privada!
    int d;
public:
    B(){
        a=1;
        b=2;
        c=3;
        d=4;
    }
};

class C:public B{
    int e;
public:
    C(){
        a = 1;
        b = 2;
        c = 3;
        d = 4;
        e = 5;
    }
};

void main(){
    B b;
    b.a = 0;
    b.b = 1;
    b.c = 2;
    b.d = 3;
}
```

Solución:

```
class A{
    int a;
protected:
    int b;
public:
    int c;
};

class B:private A{ //en este caso, la herencia es privada
    int d;
public:
    B(){
        a=1; //incorrecto
        b=2; //ok
        c=3; //ok
        d=4; //ok
    }
};

class C:public B{
    int e;
public:
    C(){
        a = 1; //incorrecto
        b = 2; //incorrecto
        c = 3; //incorrecto
        d = 4; //no, es privado de B
        e = 5; //ok
    }
};

void main(){
    B b;
    b.a = 0; //incorrecto
    b.b = 1; //no
    b.c = 2; //no
    b.d = 3; //no
}
```

11.2. Inicializador base.

El inicializador base permite especificar cómo se inicializa la clase base desde el constructor de una clase derivada. Este inicializador se utiliza en la definición, no en la declaración, del constructor. Su sintaxis general es la siguiente:

```
<id_clase_derivada>(parametros):<id_clase_base>(argumentos
   actuales), [atributos()...]{

}
```

No es necesario usar un inicializador base si la clase base cuenta con un constructor por defecto. Es importante destacar que, sin importar el orden en el que se escriban en el código, lo primero que se ejecuta al construir un objeto es el inicializador de la clase base, seguido por la inicialización de los atributos, en el orden en que fueron declarados en la clase.

Ideas importantes:

- Constructor por defecto de oficio: llama por defecto al constructor de la clase y luego en orden de declaración a los constructores por defecto (inicialización por defecto) de los atributos no estáticos. Ejemplo:

```
struct A{

};

struct B{
    Persona p{"p", "t"};
};

//llama primero a A y luego a los constructores de p y t.
```

- Si no ponemos el inicializador base en un constructor de la clase derivada, se usará el constructor por defecto de la clase base que debe existir.

```
struct A{

};

struct B:A{
    B (int c) {}
    B (float d){}
};
```

- Una clase solamente puede poner inicializador base de la clase inmediatamente anterior.
- Una clase solamente puede poner inicializadores de atributos de ella misma.
- Con el constructor de copia: si lo defino, es responsabilidad 100 % nuestra. Si no lo defino, se nos da el de oficio y el de oficio hace lo siguiente:

- Llama al constructor de copia de la clase base y luego las copias de los atributos en el orden de declaración.

- Operador de asignación: si lo defino, responsabilidad nuestra. Si no lo defino, el de oficio llama al operador igual base y después al operador igual de cada uno de los atributos en orden de declaración.

- Recordatorio del constructor por defecto de oficio: en cuanto se escribe un constructor, perdemos el constructor de oficio de por defecto. Pero la unica manera de anular el constructor de oficio de copia es escribir nuestro constructor de copia.

11.2.1. Ejemplo.

Persona.h

```
#pragma once
#include<iostream>
#include<string>
using namespace std;

class Persona{
    inline static int agno_actual{};
    int agno{}, edad{};
    string nombre, apellido, NIF;
public:
    Persona(const string &n, const string
       &ap):nombre{n},apellido{ap}{}void set_edad(int e){
        edad = e;
        agno = agno_actual - e;
    }
    void set_nif(const string &nif){
        NIF = nif;
    }
    void print(std::ostream &os){
        os<<nombre<<","<<apellido<<"("<<agno<<")";
    }
    static void set_agno_actual (int a){
        agno_actual = a;
    }
};

inline void print(const Persona &p){
    p.print(cout);
    cout<<endl;
}
```

Jugador.h:

```
#include "Persona.h"
class Jugador:public Persona{
```

```
    string equipo;
public:
    Jugador(const string &n, const string &ap, const string
       &eq):Persona(n,ap),equipo{eq}{}
    void print(ostream &os){
        Persona::print(os);
        os<<"-->"<<equipo;
    }
};
```

Alumno.h:

```
#include "Persona.h"
class Alumno:public Persona{
    int matricula;
public:
    Alumno(const string &n, const string &ap, int
       mat):Persona(n,ap),matricula(mat){}
    void print_completo (ostream &os){
        Persona::print(os);
        cout<<"--Nº"<<matricula<<endl;
    }
};
```

Main.cpp:

```
int main(){
    Jugador antonio("Antoine", "Griezmann", "Atleti");
    Alumno pepe("Pepe", "Garcia", 43826);
    antonio.set_edad(34);
    antonio.print(cout);
    cout<<endl;
    pepe.print(cout);
    print(pepe);
    print(antonio);
    antonio.Persona::print(cout);
    cout<<endl;
    pepe.print_completo(cout);
}
```

Comentarios a realizar:

- Todas las funciones que tiene persona, las tiene antonio. Luego, se ejecutará el código de la función set edad en `Persona.h`.

  ```
  antonio.set_edad(34);
  ```

- En la siguiente línea, nos damos cuenta que existen dos métodos `print` : el de `Persona` y el de `Jugador`.

  ```
  antonio.print(cout)
  ```

Los dos comparten la misma interfaz. Regla de oro: siempre se ejecuta empezando por lo más específico. Luego, `antonio` que es de tipo jugador ejecutará el de `Jugador`. Esto comportamiento se puede modificar en un futuro. Resultado de la impresión:

```
Antoine, Griezmann (1991) --> Atleti
```

- En la siguiente línea `pepe` es un `Alumno`, no tiene un `print` directamente pero tiene uno heredado. Luego, no cabe duda, ejecuta el print de `Persona`.

```
pepe.print(cout);
```

Sale por consola:

```
Pepe, Garcia (0)
```

- En la siguiente línea

```
print(pepe);
print(antonio);
```

Se trata de una función externa dentro de la clase `Persona`. Pero claro, jugador y alumno son ambas personas. A efectos prácticos, la función solamente ve la faceta `Persona` del objeto recibido. Imprime por consola:

```
Pepe, Garcia (0)
Antoine, Griezamnn (1991)// "antonio ejecuta print como
   persona"
```

- Siempre puedo especificar un ámbito de los que tengo heredados. La siguiente línea se entiende literalmente como: "el `print` que vería si considero que es desde `Persona`".

```
antonio.Persona::print(cout);
```

Imprime por consola:

```
Antoine, Griezmann (1991)
```

- Sobre la linea:

```
pepe.print_completo(cout); //da error? a revisar
```

Ideas adicionales:

- Regla importante: cuando en una clase se define una clase o un atributo con un identificador, automáticamente en ese nivel, se anulan todos los prints de las clases anteriores.
- Una persona puede ser apuntada por un puntero, luego, un puntero a `Persona` puede recibir una dirección de pepe porque pepe es una persona.

```
Persona *p = &pepe;
```

Lo que ocurre es que solamente se tendrá acceso a lo que se ve como `Persona`.

```
p->set_agno()... //ok
p->print //ok
p->print_completo() //da error
```

- Lo mismo va a ocurrir con las referencias.

```
Persona &j = pepe;
```

Desde j solamente se pueden ejecutar los métodos públicos que se ven por ser `Persona`. Luego, de pepe no puedo ver su `equipo` (`equipo` es una variable privada). Por otro lado, si se escribe:

```
Persona &j =n antonio;
```

La siguiente línea:

```
j.print(cout);
```

j es una variable de tipo `Persona`, no jugador, luego se realiza lo que imprimiría antonio como `Persona`.

11.3. Ejemplo de examen.

Se tiene el siguiente código:

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

struct A {
    A() { cout << "A()" << endl; }
    A(const A& a) { cout << "A_COPY" << endl; }
    A& operator = (const A& a) {
        cout << "A_EQUAL" << endl;
        return *this;
    }
};

//nueva clase:
struct B :A {};

struct C :A {
    C() { cout << "C()" << endl; }
    C(const C& c) { cout << "C_COPY" << endl; }
    C& operator = (const C& c) {
        cout << "C_EQUAL" << endl;
        return *this;
    }
};

//el main
int main() {
    B b1, b2(b1);
    b1 = b2;
    C c1, c2(c1);
    c1 = c2;
    return 0;
}
```

El `main` se puede reorganizar como:

```
int main() {
    B b1,      /*(1)*/
    b2(b1);    /*(2)*/
    b1 = b2;   /*(3)*/
    C c1,      /*(4)*/
    c2(c1);    /*(5)*/
    c1 = c2;   /*(6)*/
    return 0;
}
```

Comentarios a realizar:

- Línea (1). Se construye el objeto `b1`, que es un objeto de tipo `B`. Como `B` hereda de `A` y no tiene constructor propio, se llama al constructor por defecto de `A`. Salida por consola:

```
A()
```

- Línea (2). Se construye `b2` a partir de `b1`, usando el constructor de copia heredado desde `A`. Salida por consola:

```
A_COPY
```

- Línea (3). Se realiza una asignación entre dos objetos `B`. Se usa el operador de asignación de `A` porque `B` no tiene uno propio. Salida por consola:

```
A_EQUAL
```

- Línea (4). Se construye `c1`. Primero se llama al constructor por defecto de `A` (porque `C` hereda de `A` y la clase base siempre tiene que ser inicializada) y luego al de `C`. Salida por consola:

```
A()
C()
```

- Línea (5). Se construye `c2` a partir de `c1`. Primero se llama al constructor de copia de `A` (por herencia), luego al de `C`. Salida por consola:

```
A()
C_COPY
```

- Línea (6). Se hace una asignación entre `c1` y `c2`. Salida por consola:

```
C_EQUAL
```

Salida total por consola:

```
A()
A_COPY
A_EQUAL
A()
C()
A()
C_COPY
C_EQUAL
```

Modifico el código:

```
struct C :A {
    C() { cout << "C()" << endl; }
    C(const C& c):A(c) { cout << "C_COPY" << endl; }
    C& operator = (const C& c) {
        cout << "C_EQUAL" << endl;
        return *this;
    }
};
```

Entonces, las salidas por consola se mantienen iguales con respecto al caso anterior a excepción de la línea (5).

```
int main() {
    B b1,      /*(1)*/
    b2(b1);    /*(2)*/
    b1 = b2;   /*(3)*/
    C c1,      /*(4)*/
    c2(c1);    /*(5)*/
    c1 = c2;   /*(6)*/
    return 0;
}
```

El constructor de copia de `C` invoca explícitamente al constructor de copia de la clase base `A` mediante la sintaxis `: A(c)`. Esto asegura que la subestructura heredada desde `A` se copie correctamente. Luego se ejecuta el cuerpo del constructor de copia de `C`. Salida por consola:

```
A_COPY
C_COPY
```

Salida total por consola:

```
A()
A_COPY
A_EQUAL
A()
C()
A_COPY
C_COPY
C_EQUAL
```

11.4. Herencia Múltiple.

La herencia múltiple es un mecanismo que permite a una clase derivar simultáneamente de varias clases base. Esto ofrece una gran flexibilidad y potencia en el diseño de sistemas, ya que permite combinar comportamientos y características de distintas clases en una sola. En términos de jerarquía, posibilita estructuras más complejas, como copas de árboles, en lugar de simples ramas lineales.

Es importante tener en cuenta que, cuando se hereda de múltiples clases, existe un orden de resolución de atributos y métodos. Este orden se determina por el orden en que se declaran las clases base en la definición de la subclase. En lenguajes como Python, este orden se gestiona a través del algoritmo conocido como MRO (Method Resolution Order).

11.4.1. Ejemplo. Ferrari

```
class ConMarca{
    string marca;
public:
    ConMarca(const string &m):marca(m){}
    void print(){cout<<marca<<"(TM)";
};

class Coche{
    string modelo;
    int potencia;
public:
    Coche(const string &m, int pot):modelo(m), potencia(pot){}
    void print(){cout<<modelo<<"("<<potencia<<"cv)";}
};

class F50:public ConMarca, public Coche{
    int num_serie{};
public:
    F50(int num): ConMarca("Ferrari"),Coche("F50", 850),
       num_serie(num){}
};

void main(){
    F50 mi_ferrari(22731);
    mi_ferrari.print(); // ¿Qué ocurre aquí?
}
```

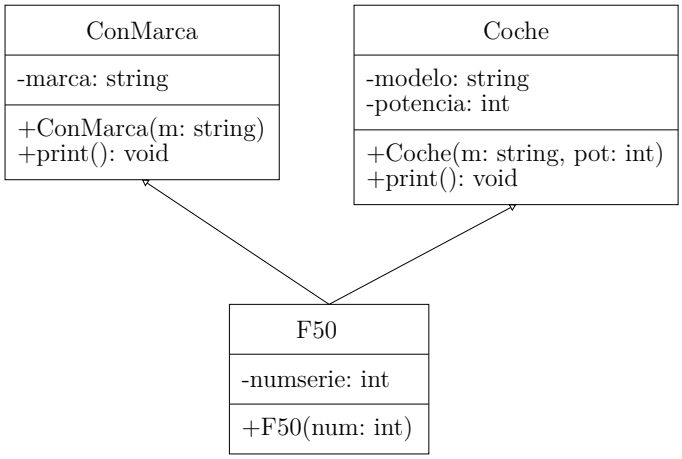


Figura 2: Diagrama UML.

La clase `F50` hereda tanto de `ConMarca` como de `Coche`, lo que refleja que un `F50` es un objeto con marca *y* es un coche. Sin embargo, este diseño introduce un problema:

- El objeto `mi_ferrari` tiene dos funciones `print()`, una heredada de `ConMarca` y otra de `Coche`, por lo tanto la llamada:

```
mi_ferrari.print();
```

  genera un error de **ambigüedad**, ya que el compilador no sabe a cuál de las dos funciones referirse.

- Para solucionarlo, se puede especificar el ámbito:

```
mi_ferrari.ConMarca::print();
mi_ferrari.Coche::print();
```

- Alternativamente, se puede definir un nuevo método `print() en F50` que combine ambos:

```
class F50:public ConMarca, public Coche{
    int num_serie{};
public:
    F50(int num): ConMarca("Ferrari"),Coche("F50", 850),
       num_serie(num){}

    void print(){
        Coche::print();
        cout<<"Producto de ";
        ConMarca::print();
        cout<<" num"<< num_serie;
    }
};
```

  Ahora, la llamada:

```
mi_ferrari.print();
```

funciona correctamente, ejecutando el método `print()` de `F50`, que resuelve la ambigüedad de forma explícita.

Por otro lado, cuando se trabaja con herencia, es común acceder al objeto derivado a través de referencias o punteros a la clase base. En este caso, solo se tiene acceso a los miembros de la parte de la clase correspondiente.

```
void main() {
    F50 mi_ferrari(22731);
    mi_ferrari.print(); // llamado a F50::print()

    ConMarca &r = mi_ferrari;
    r.print(); // llamado a ConMarca::print(), solo ve esa parte

    ConMarca *p = &mi_ferrari;
    p->print(); // también llamado a ConMarca::print()
}
```

En este ejemplo, la referencia y el puntero a `ConMarca` sólo pueden acceder a lo definido en `ConMarca`, no a lo de `F50` ni `Coche`.

11.5. Herencia Virtual.

A veces aparece un problema adicional, y es la redundancia, el tener un mismo objeto declarado dos veces a la vez donde pueden haber incoherencias. La herencia virtual es un mecanismo que nos evita esta duplicidad, el `virtual` significa, de una manera, poner en conjunto las partes virtuales.

Ejemplo de jugador español:

```
class Espagnol: virtual public Persona{

};

class Futbolista: virtual public Persona{

};

class JugadorEspagnol: public Futbolista, public Español{

};
```

11.6. Ejemplo de clase. Conversión up_cast.

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

class Guerrero {
    int vida = 10;
public:
    int get_vida() const { return vida; }
    void pelea() { cout << endl << "Peleo"; } // (1)
    void print() const { cout << endl << "Guerrero(" << vida <<
       ")"; }
};

class Mago : public Guerrero {  // (2)
public:
    void hechiza() { cout << endl << "Hechizo"; } // (3)
    void print() const { cout << endl << "Mago y ";
       Guerrero::print(); }
};

class Clerigo : Guerrero {
public:
    void bendice() { cout << endl << "Bendigo"; }
    void print() const { cout << endl << "Clerigo(" <<
       get_vida() << ")"; }
};

void imprime(const Guerrero &g) {// (4)
    g.print();
}

int main() {
    Guerrero g;
    Mago m;
    Clerigo c;

    g.pelea();      // (5)
    m.hechiza();    // (6)
    m.pelea();      // (7)
    m.print();      // (8)
    c.print();      // (9)

    Guerrero g2 = m, *pg = &m; // (10)(11)
    Mago* pm;
    imprime(g);      // (12)
    imprime(m);      // (13)

    pg->print();     // (14)
```

```
    pm = static_cast<Mago*>(pg); // (15)
    pm->print();     // (16)

    pg = &g;
    pg->print();     // (17)
    pm = static_cast<Mago*>(pg); // (18)
    pm->print();

    // imprime(c);              // ERROR
    // Guerrero g3 = c;         // ERROR
    // Guerrero pg3 = &c;       // ERROR
}
```

Comentarios a realizar

- Línea (1). Todos los guerreras saben pelear.
- Línea (2). Los magos también son guerreros.
- Línea (3). Solos los magos lanzan hechizos.
- Línea (4). Se puede porque la clase base es `public`.
- Línea (5). Llamada directa a un método no virtual desde un objeto de tipo `Guerrero`. Se realiza **enlace estático**, por lo tanto se imprime:

  ```
  Peleo
  ```

- Línea (6).

  ```
  m.hechiza();
  ```

  Llamada directa al método propio de `Mago`. Enlace estático:

  ```
  Hechizo
  ```

- Línea (7).

  ```
  m.pelea();
  ```

  `Mago` hereda públicamente de `Guerrero`, por lo que puede usar su método `pelea`. Enlace estático:

  ```
  Peleo
  ```

- Línea (8).

  ```
  m.print();
  ```

  Se ejecuta el método `print` definido en `Mago`, que a su vez llama al método `Guerrero::print`. Se imprime:

```
Mago y Guerrero(9)
```

- Línea (9).

```
c.print();
```

`Clerigo` hereda privadamente de `Guerrero`, pero al estar en una clase amiga (el propio método), puede acceder a `get_vida`. Se imprime:

```
Clerigo(10)
```

- Líneas (10)-(11).

```
Guerrero g2 = m;
Guerrero *pg = &m;
```

`g2` es una copia (slicing) de la parte `Guerrero` de `m`, y `pg` es un puntero a la parte base al ser de tipo `Guerrero`. No hay polimorfismo porque los métodos no son virtuales.

- Línea (12).

```
imprime(g);
```

Llamada al método `print` de `Guerrero` mediante referencia. Se imprime:

```
Guerrero(10)
```

- Línea (13).

```
imprime(m);
```

Se pasa un objeto `Mago` como referencia a `Guerrero`. No hay uso del método de `Mago` porque `print` no es virtual. Se imprime:

```
Guerrero(10)
```

- Línea (14).

```
pg->print();
```

Esta línea tiene relación con las líneas (6) y (7). El puntero es de tipo `Guerrero*`, y como `print` no es virtual, se usa el tipo estático. Se imprime:

```
Guerrero(10)
```

- Línea (15)-(16).

```
pm = static_cast<Mago*>(pg);
pm->print();
```

Conversión segura ya que originalmente `pg` apuntaba a un `Mago`. Se imprime:

```
Mago y Guerrero(10)
```

- Línea (17).

```
pg = &g;
pg->print();
```

Ahora `pg` apunta a un `Guerrero`. Llamada normal:

```
Guerrero(10)
```

- Línea (18).

```
pm = static_cast<Mago*>(pg);
pm->print();
```

Conversión no segura. Se fuerza el compilador a tratar un `Guerrero` como si fuera un `Mago`, lo cual es incorrecto. El comportamiento es indefinido. Pero el compilador lo permite:

```
Mago y Guerrero(10) // ¡Peligroso!
```

- Últimas líneas comentadas:

```
imprime(c);             // ERROR
Guerrero g3 = c;        // ERROR
Guerrero *pg3 = &c;     // ERROR
```

Estas líneas no compilan porque `Clerigo` hereda `privadamente` de `Guerrero`. Por tanto, no se puede acceder a su parte pública desde fuera de la clase `Clerigo`.

12. Sesión 12. Polimorfismo.

12.1. Concepto.

El polimorfismo se fundamenta en el uso de métodos virtuales. Una clase es polimórfica cuando tiene un método virtual.

12.2. Métodos virtuales.

Un método `virtual` es una función miembro declarada en una clase base, que puede ser redefinida por las clases derivadas. Esto permite que, incluso cuando un objeto es accedido a través de un puntero o una referencia de la clase base, se ejecute la versión del método definida en la clase derivada, utilizando los datos específicos del objeto derivado.

La sintaxis de un método `virtual` se declara en la clase base con la palabra clave `virtual`:

```
virtual <tipo> <id_metodo>(<parametros>);
```

Dependiendo del uso, puede tomar distintas formas:

- Solo declaración (sin definición en línea):

  ```
  virtual <tipo> <id_metodo>(<parametros>);
  ```

  Esta forma solo declara la función como virtual, y su definición se hace posteriormente fuera de la clase.

- Declaración y definición en línea (dentro de la clase):

  ```
  virtual <tipo> <id_metodo>(<parametros>) {
      // código
  }
  ```

  Aquí se declara y define la función virtual directamente dentro de la clase.

- Función virtual pura:

  ```
  virtual <tipo> <id_metodo>(<parametros>) = 0;
  ```

  Esto declara una función `virtual` **pura**, lo que convierte a la clase en una **clase abstracta**. Las clases derivadas están obligadas a sobrescribir esta función.

- Definición fuera de la clase:

  ```
  <tipo> <nombre_clase>::<id_metodo>(<parametros>) {
      // código
  }
  ```

  Si la función virtual fue solo declarada en la clase, se puede definir fuera de ella usando el operador de resolución de ámbito `::`.

Ideas clave:

- Una vez declarado un método como `virtual`, lo seguirá siendo `virtual` en las clases derivadas. Es decir, la propiedad `virtual` se hereda.
- La virtualidad funciona si y solo si, el método se llama igual, si tiene el mismo tipo y número de argumentos y el mismo valor retorno. Es decir, la equivalencia debe de ser absoluta. Por ese motivo, en C++11 se añade la palabra clave `override`, se incluye en la declaración de la función virtual que quiere sobreescribir.

```
virtual <tipo> <id_metodo> (<parametros>) override ;
   {<codigo>} =0;
```

  `override` es un mecanismo de seguridad: si no se está produciendo una redefinición en un método heredado, entonces da error, es recomendable su uso.

  También se puede añadir la palabra clave `final`, en cualquier orden:

```
virtual <tipo> <id_metodo> (<parametros>) override final ;
   {<codigo>} =0;
virtual <tipo> <id_metodo> (<parametros>) final override ;
   {<codigo>} =0;
```

  `final` sirve para declarar que a partir de ese instante, nadie más va a poder sobrecargar el método. Rompe el mecanismo de virtualidad.

- El nivel de acceso no afecta a la virtualidad. Ejemplo:

```
//se tiene una clase A
struct A{
    virtual void print(){}
};

//se tiene una clase B heredada de A
struct B:A{
private:
    void print()override{}
}

void main(){
    B = b;
    A &r = b;
    r.print();
    b.print();
}
```

  Comentarios a realizar:

  - En la siguiente línea, qué método `print` se ejecuta?

```
r.print();
```

`r` es una referencia de tipo `A` a un objeto `b` de tipo `B`, luego, se realiza una llamada inicial al método `print` de la clase `A`:

```
virtual void print(){}
```

Sin embargo al ser `virtual`, pregunta a `b`: ¿Pero tú qué eres realmente? `b` es un objeto de tipo `B` luego, le dice: ejecuta entonces, tu propio método `print`. Por lo tanto, se ejecuta:

```
B.print;
```

- La siguiente línea da error por la privacidad del método:

```
b.print();
```

- La llamada a un método `virtual` se resuelve en tiempo de ejecución, es decir, siempre en función del tipo de objeto referenciado o apuntado. Es un enlace dinámico.
- La llamada de un método normal (no `virtual`) se resuelve siempre en función del tipo de la referencia o el puntero utilizado.
- Una llamada a un método `virtual` específico exige el uso del operador `scope` `::` el cual rompe el mecanismo de virtualidad.

```
<clase>::<metodo>
```

Ejemplo:

```
#include <iostream>

struct A {
    virtual void print() { std::cout << "A::print\n"; }
};

struct B : public A {
    void print() override { std::cout << "B::print\n"; }
};

int main() {
    B b;
    A& a = b;
    a.print(); // Llama a B::print() por virtualidad
    a.A::print(); // Fuerza la llamada a A::print(), rompe
       la virtualidad
    b.print(); // Llama a B::print()
    b.A::print(); // Fuerza la llamada a A::print()
}
```

- Los métodos virtuales son un poco más lentos en la ejecución.

- No se debe de llamar nunca a funciones virtuales puras ni en constructores ni destructores porque su comportamiento no está definido.

Sobre los constructores y destructores:

- No existe el concepto de constructor `virtual`. Sin embargo, nos interesa que un objeto se duplique manteniendo su naturaleza polimórfica. ¿Cómo hacerlo? Clonando, se usa un método que tiene este prototipo:

```
virtual <clase_base> *clone() = 0;
```

- Los destructores sí pueden ser virtuales y deben serlo en el caso de que las clases derivadas tengan que liberar o hacer operaciones de destrucción. Ejemplo:

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

class A{
public:
    int identif;
    ~A(){cout<<"destruye A\n";}
};

class B:public A{
public:
    int *valores = new int[50];
    ~B(){delete[] valores;
        cout<<"destruye B\n";
    }
};

int main(){
    A *c = new B; //Se crea un objeto de tipo B, pero se
       guarda en un puntero a A
    delete c; //imprime "destruye A"
}
```

Si añadimos `virtual`:

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

class A{
public:
    int identif;
    virtual ~A(){cout<<"destruye A\n";}
};

class B:public A{
public:
    int *valores = new int[50];
```

```
    ~B(){delete[] valores;
        cout<<"destruye B\n";
    }
};

int main(){
    A *c = new B;
    delete c; // Gracias al destructor virtual de A, se
       llama ~B() y luego ~A()
}
```

12.3. Ejemplo. Polígonos.

Ejemplo clásico de clase. Se tiene una clase `Poligono` que tiene una interfaz y después tenemos unos métodos `perimetro`, `area` y el `print`. Heredamos de `Poligono` `Triangulo`.

```
class Poligono{
protected:
    int num_lados, lado;
public:
    Poligono(int n, int l):num_lados(n), lado(l){}
    int perimetro(){return num_lados *lado;}
    double area(){return 1.0;}
    void print(){cout<<"Poligono de "<<num_lados<<" y lado
       "<<lado<<" y area "<<area();
    }
};

struct Triangulo:Poligono{
    Triangulo(int l):Poligono(3,l){}//recordamos:
       obligatoriamente, al haber constructor por defecto, la
       clase base debe ser inicializada
    double area(){return 0,433*lado*lado;}
    void print(){
        cout<<"Triangulo de lado "<<lado;
    }
};

struct Cuadrado:Poligono{
    Cuadrado(int l):Poligono(4,l){}
    double area(){return lado*lado;}
    void print(){cout<<"Cuadrado de lado "<<lado;
    }
};

void print(Poligono p){
    p.print();
}

void print2(Poligono &p){
    p.print();
}

void main(){
    Poligono p(5,2);
    Triangulo t(4);
    Cuadrado c(8);
}
```

Entonces, si escribimos en el main:

```
void main(){
    Poligono p(5,2);
    Triangulo t(4);
    Cuadrado c(8);
    p.print();      /*(1)*/
    t.print();      /*(2)*/
    Poligono p2 = t;
    p2.print();     /*(3)*/
    Poligono *pp = &t;
    pp->print();    /*(4)*/
    Poligono &rp = t;
    rp.print();     /*(5)*/
}
```

Comentarios a realizar:

- Línea (1). Llamada directa a `print()` desde un objeto de tipo `Poligono`, sin ambigüedad. Se ejecuta:

  ```
  Poligono de 5 y lado 2 y area 1
  ```

- Línea (2). Llamada directa desde un objeto de tipo `Triangulo`. Se ejecuta el método `print()` de la clase `Triangulo`:

  ```
  Triangulo de lado 4
  ```

- Línea (3). Aquí se crea un nuevo objeto `Poligono` a partir de un objeto `Triangulo`. Ocurre **slice** (recorte): se copia solo la parte base de `t`, perdiendo toda la información adicional de `Triangulo`. Luego, al llamar `p2.print()`, se ejecuta el método de la clase `Poligono`:

  ```
  Poligono de 3 y lado 4 y area 1
  ```

- Línea (4). Se crea un puntero de tipo `Poligono*` apuntando a un `Triangulo`. Sin embargo, como `print()` no es `virtual`, no hay polimorfismo en tiempo de ejecución. Se ejecuta el método `print()` de `Poligono`, no el de `Triangulo`:

  ```
  Poligono de 3 y lado 4 y area 1
  ```

- Línea (5). Similar al caso anterior, se tiene una referencia de tipo base a un objeto derivado. Pero, nuevamente, al no ser `print()` `virtual`, se llama al método según el tipo estático de la referencia (`Poligono`), no al tipo dinámico (`Triangulo`):

  ```
  Poligono de 3 y lado 4 y area 1
  ```

Modificación 1: añadir virtualidad.

```
class Poligono{
protected:
    int num_lados, lado;
public:
    Poligono(int n, int l):num_lados(n), lado(l){}
    int perimetro(){return num_lados *lado;}
    double area(){return 1.0;}
    virtual void print(){cout<<"Poligono de "<<num_lados<<" y
       lado "<<lado<<" y area "<<area();
    }
};
```

A partir de ahora, la función `print` pregunta "tu realmente quién eres?" Entonces:

```
void main(){
    Poligono p(5,2);
    Triangulo t(4);
    Cuadrado c(8);
    p.print();            /*(1)*/
    t.print();            /*(2)*/
    Poligono p2 = t;
    p2.print();           /*(3)*/
    Poligono *pp = &t;
    pp->print();          /*(4)*/
    Poligono &rp = t;
    rp.print();           /*(5)*/
    c.Poligono::print(); /*(6)*/
}
```

Comentarios a realizar:

- Línea (1). Llamada directa a `print()` desde un objeto de tipo `Poligono`. Se ejecuta sin ambigüedad:

  ```
  Poligono de 5 y lado 2 y area 1
  ```

- Línea (2). Llamada directa a `print()` desde un objeto de tipo `Triangulo`. Se ejecuta el método sobrescrito de `Triangulo`, como antes:

  ```
  Triangulo de lado 4
  ```

- Línea (3). Se crea un nuevo objeto `Poligono` a partir de un `Triangulo`. Ocurre **object slicing**, es decir, solo se conserva la parte base del objeto. La llamada a `p2.print()` invoca la versión de `Poligono`:

  ```
  Poligono de 3 y lado 4 y area 1
  ```

- Línea (4). Se tiene un puntero de tipo `Poligono*` apuntando a un objeto `Triangulo`. Como ahora `print()` es `virtual`, el sistema pregunta dinámicamente "¿quién eres?", y al saber que es un `Triangulo`, se ejecuta el método sobrescrito:

```
Triangulo de lado 4
```

- Línea (5). Caso similar al anterior, pero usando una referencia en lugar de un puntero. Al ser referencia a base y método `virtual`, se ejecuta también el método sobrescrito:

```
Triangulo de lado 4
```

- Línea (6). Aquí se utiliza el operador de resolución de ámbito `::` para invocar explícitamente el método `print()` de la clase `Poligono`, incluso desde un objeto de tipo `Cuadrado`. Esto rompe el mecanismo de virtualidad, y se ejecuta directamente el método de la clase base:

```
Poligono de 4 y lado 8 y area 1
```

Modificación 2: otra virtualidad más.

```
class Poligono{
protected:
    int num_lados, lado;
public:
    Poligono(int n, int l):num_lados(n), lado(l){}
    int perimetro(){return num_lados *lado;}
    virtual double area(){return 1.0;}
    virtual void print(){cout<<"Poligono de "<<num_lados<<" y
       lado "<<lado<<" y area "<<area();
    }
};
```

Entonces:

```
void main(){
    Poligono p(5,2);
    Triangulo t(4);
    Cuadrado c(8);
    p.print();            /*(1)*/
    t.print();            /*(2)*/
    Poligono p2 = t;
    p2.print();           /*(3)*/
    Poligono *pp = &t;
    pp->print();          /*(4)*/
    Poligono &rp = t;
    rp.print();           /*(5)*/
    c.Poligono::print(); /*(6)*/
}
```

Comentarios a realizar:

- Línea (1). Llamada directa a `print()` desde un objeto de tipo `Poligono`. Se ejecuta sin ambigüedad, usando la versión de `Poligono`, y también llama a `area()` de `Poligono`:

```
Poligono de 5 y lado 2 y area 1
```

- Línea (2). Llamada directa a `print()` desde un objeto de tipo `Triangulo`. Se ejecuta el método sobrescrito en `Triangulo`:

```
Triangulo de lado 4
```

- Línea (3). Se crea un nuevo objeto `Poligono` a partir de un `Triangulo`, lo que provoca **object slicing**. Se copia solo la parte base del objeto derivado. Por tanto, al ejecutar `p2.print()`, se ejecuta el método de `Poligono` y su `area()`:

```
Poligono de 3 y lado 4 y area 1
```

- Línea (4). Se usa un puntero de tipo `Poligono*` apuntando a un objeto `Triangulo`. Como `print()` y `area()` son virtuales, se invocan dinámicamente según el tipo real del objeto. Se ejecuta el `print()` de `Triangulo`, sin llamar a `Poligono::print()` ni a `Poligono::area()`:

```
Triangulo de lado 4
```

- Línea (5). Caso similar al anterior, pero usando una referencia. Nuevamente se activa el polimorfismo y se llama a `Triangulo::print()`:

```
Triangulo de lado 4
```

- Línea (6). Se llama directamente a `Poligono::print()` desde un objeto `Cuadrado` usando el operador `::`, lo que desactiva el polimorfismo. Sin embargo, dentro de `Poligono::print()`, se hace una llamada a la función virtual `area()`, y ahí sí se recupera el comportamiento dinámico. Se ejecuta `Cuadrado::area()`, aunque se esté dentro del contexto estático de `Poligono`:

```
Poligono de 4 y lado 8 y area 64
```

Por otro lado, tras las iteraciones que hemos realizado:

- En cuanto al siguiente método:

```
void print(Poligono p){
    p.print();
}
```

  Hace impresión como `Poligono`, imprime `Poligono de ... area 1`

- En cuanto al siguiente método:

```
void print2(Poligono &p){
    p.print();
}
```

  Se escribe una referencia a un objeto. La referencia limita a que las únicas funciones a ejecutar son solo como `Poligono`.

12.4. Funciones virtuales puras y clases abstractas.

Recordamos del apartado anterior, las funciones virtuales puras toman la forma:

```
virtual <tipo> <id_metodo>(<parametros>) = 0;
```

Una clase que tenga una función virtual pura sin definir, se denomina clase abstracta.

En las clases abstractas:

- No se pueden crear objetos de esa clase.
- Sí se pueden usar punteros y referencias.
- Si una clase derivada no define alguno de los métodos virtuales puros, pasa a ser abstracta.

12.5. Examen de Laboratorio B Julio 2017.

12.5.1. Enunciado.

Desarrollar las siguientes clases de C++:

- **Volumen**: Clase base abstracta interfaz que contiene el método virtual puro `double comp_vol()` para obtener el volumen de una figura. Además, sobrecarga adecuadamente el operador texto (`<<`) (como función independiente, complementada por la función miembro virtual `print`).
- **Esfera**: Clase derivada de `Volumen` determinada por el radio de una esfera.
- **Cilindro**: Clase derivada de `Volumen` determinada por la altura y volumen de un cilindro.

El código cliente que valida el desarrollo de las clases anteriores es una aplicación de *Consola* con el siguiente programa:

```
#include <iostream>
using namespace std;
#define MAX_NUM 3

void main(){
    Volumen* fig[2];
    fig[0]= new Esfera(3.0);
    fig[1]= new Cilindro(3.0 /* r */, 5.0 /* h*/);
    cout<<fig[0]<<endl; /*IMPRIMIRA: Esfera de radio 3.0*/
    cout<<"El volumen de la esfera es: "<<fig[0]->comp_vol()<<endl;/*
       113.04*/
    cout<<"El volumen del cilindro es: "<<fig[1]->comp_vol()<<endl;/*
       141.3*/
}
```

Nota: Se evaluará el correcto uso de los conceptos de POO (encapsulamiento, herencia, polimorfismo, sobrecarga, …) no dando por bueno soluciones que no los utilicen.

12.5.2. Resolución.

Comentarios iniciales sobre el código.

```
fig[0]= new Esfera(3.0); //genera Esfera de radio 3
fig[1]= new Cilindro(3.0, 5.0); //genera Cilindro de radio 3 y
   altura 5
```

Entonces, escribo:

```
struct Volumen{
    virtual double comp_vol() = 0;
    virtual void print(ostream &os) = 0;
};
```

Por otro lado, sobrecarga del operador:

```
ostream & operator<<(ostream &os, Volumen *vol){
    vol->print(os);
    return os;
}
```

En cuanto la clase `Esfera`:

```
class Esfera: public Volumen{
    double radio{};
public:
    Esfera(double r): double(r){}
    double comp_vol()override{
        return (4.0/3.0)*r*r*r*3.14159;
    }
    void print(ostream &os)override{
        os<<"Esfera de radio"<<radio;
    }
};
```

12.6. Clonado Polimórfico.

Se recomienda ver como introducción a esta sección la parte de 12.2.

Una limitación de C++ es que no existe el concepto de constructor virtual. Esto impide crear objetos polimórficamente usando constructores. Sin embargo, es habitual querer duplicar un objeto respetando su tipo dinámico. Para ello, se utiliza el patrón de clonado polimórfico, mediante el uso de un método `virtual`.

12.6.1. Patrón 1.

```
class Base {
public:
    virtual Base* clonar() const = 0;
    virtual ~Base() {}
};
```

```
class Derivada : public Base {
public:
    Base* clonar() const {
        return new Derivada(*this);
    }
};
```

12.6.2. Patrón 2.

También es útil definir una forma de imprimir objetos polimórficos mediante el operador `<<`. Para ello, se define una función `print virtual`.

```
class Base {
public:
    virtual ostream &print(ostream &os) const = 0;
    virtual ~Base() {}

    inline ostream &operator<<(ostream &os, const Base &obj) {
        return obj.print(os);
    }
};
```

```
class Derivada : public Base {
public:
    ostream &print(ostream &os) const {
        return os;
    }
};
```

12.7. Examen de Laboratorio.

Tenemos una galleta que deriva de producto y es una clase abstracta, `main.cpp` es:

```
int main(){
    Carrito mi_carrito;
    while(char a = cin.get()){
        if(a == 'g') mi_carrito += new Galleta;
        if(a == 'l') mi_carrito += new Leche;
        if(a == 'f') break;
    }
    mi_carrito.print();
    Carrito copia = mi_carrito;
    Copia.print();
}
```

Tras pulsar g l f :

```
el carrito tiene:
Galleta Maria a 2.60
1 litro de leche a 1.20
el total a pagar es 3.80
```

Solución:

```
class Carrito{
    vector <Producto*> lista;
public:
    Carrito &operator+=(Producto*p){
        lista.push_back(p);
        return *this;
    }
    void print(){
        cout<<"El carrito tiene:";
        for(auto p:lista) p->print();
        cout<<"El total a pagar es:"<<total();
    }
    double total(){
        double suma{};
        for(auto p:lista) suma += p->precio();
        return suma;
    }
    ~Carrito(){
        for(auto p:lista) delete p;
    }
    Carrito (const Carrito &);
}
```

Por otro lado:

```
class Producto{
    double precio;
public:
    double precio(){return precio;}
    virtual void print() const = 0;
    virtual Producto *clonar() const = 0;
    Producto (double p):precio(p<= 0.1? 0.1:p){}
};
```

13. Sesión 13. Templates.

13.1. Introducción.

Una plantilla es un mecanismo de C++ que permite definir funciones o clases genéricas que trabajan con diferentes tipos de datos. Se utiliza mediante una declaración previa con la siguiente sintaxis:

```
template<class T> // T se define como un tipo de dato genérico
```

Ejemplo: plantilla para la función `swap`, que intercambia dos variables:

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

Gracias a esta plantilla, podemos usar `swap` con distintos tipos de datos. Por ejemplo:

```
int main() {
    int x = 8, y = 9;
    double xx = 18.0, yy = 9.8;
    swap(x, y);
    swap(xx, yy);
    swap<int>(x, y);
}
```

Comentarios a realizar:

- En la siguiente línea, `T` se deduce automáticamente como `int`.

  ```
  swap(x, y);
  ```

- En la siguiente línea, `T` se deduce automáticamente como `double`.

  ```
  swap(xx, yy);
  ```

- También es posible especificar el tipo de datos que se recibe mediante los caracteres `<>`:

  ```
  swap<int>(x, y);
  ```

  Esta línea es equivalente a la línea:

  ```
  swap(x, y);
  ```

  Este mecanismo se conoce como instanciación explícita de plantillas. El compilador genera una versión específica de la función `swap` para cada tipo necesario.

  Con las clases genéricas también se usa la misma idea. Por ejemplo, la STL define un vector genérico:

  ```
  vector<int> v; //el tipo de datos variable es un entero
  ```

13.2. Plantillas con parámetros múltiples y valores constantes.

Las plantillas en C++ pueden incluir no solo tipos genéricos, sino también parámetros que no son tipos, como constantes (`int`, `bool`, etc.).

La función siguiente lleva una cuenta de cuántas veces se ha invocado.

```
template<class T>
    T* construir() {
    static int cuenta = 0;
    cuenta++;
    return new T;
}
```

Ejemplo de uso:

```
void* p = construir<int>();//Se instancia para el tipo int
```

En este ejemplo, `cuenta` es una variable estática local, por lo que se mantiene entre llamadas a la función, pero es independiente para cada tipo `T` con el que se instancie la plantilla.

13.3. Plantillas con múltiples parámetros de tipo.

Las plantillas también pueden aceptar varios parámetros de tipo:

```
template <class A, class B, class C>
void func(B b, C c, int i);
```

Si escribimos entonces:

```
int i = 8;
func(b, c, i) = 8;
func<A, B, C>(b, c, i);
func<A>(b, c, i);
func<A>(b, c);
```

Comentarios a realizar:

- `func(b, c, i);` no compila porque el compilador no puede deducir el tipo de `A` a partir de los argumentos.
- `func<A, B, C>(b, c, i);` es la forma completa, especificando los tres tipos explícitamente.
- `func<A>(b, c, i);` también puede funcionar si `B` y `C` pueden deducirse del contexto.
- `func<A>(b, c);` no es válida, ya que falta el argumento entero que espera la función (`int i`).

13.4. Plantillas con parámetros no tipo.

También es posible usar valores como parámetros de plantilla. Por ejemplo:

```
template <class T, int n>
void print(T* t) {
    for(int i = 0; i < n; i++)
        cout << t[i];
}
```

Aquí, `n` es un parámetro constante que indica el número de elementos que se deben imprimir desde el arreglo `t`. Este valor debe conocerse en tiempo de compilación.

Ejemplo de uso:

```
int datos[] = {1, 2, 3, 4};
print<int, 4>(datos);
```

Al compilar, nos aparece por consola:

```
1234
```

13.4.1. Ejemplo.

Se tiene el siguiente código:

```
#include <iostream>
class Vector {
public:
    float x, y;
    bool operator>(Vector v) {
        return ((x * x + y * y) > (v.x * v.x + v.y * v.y)) ?
           true : false;
    }
};

template <class T> T max(T a, T b) {
    return (a > b) ? a : b; //operador ternario
}

void main() {
    Vector v1 = { 2, 3 }, v2 = { 1, 5 };
    int x = 2, y = 3;
    std::cout << "Mayor: " << max(x, y) << std::endl;
    std::cout << "Mayor: " << max(v1, v2).x << "," << max(v1,
       v2).y << std::endl;
}
```

Al compilar nos aparece por consola:

```
Mayor: 3
Mayor: 1,5
```

14. Ejemplos de estudio.

14.1. Ejemplo 1.

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

class coordinador {
public:
    int curso;
    string nombre;
    coordinador() {
        cout << "Constructor por defecto coordinador" << endl;
    }

    coordinador(const string& name, int count):nombre(name),
       curso(count) {
        cout << "Este es el principal " << nombre << " de " <<
           curso << " curso" << endl;
    }

    ~coordinador() {
        cout << "Elimina el coordinador" << endl;
    }
};

class subdelegado : public coordinador {
public:
    subdelegado() {
        cout << "Constructor por defecto subdelegado" << endl;
    }
    subdelegado(const string& name, int count) :
       coordinador(name, count){
        cout << "Este es el subdelegado " << nombre << " de "
           << curso << " curso" << endl;
    }
};

int main() {
    coordinador c1;
    coordinador c2("Laura", 3);
    subdelegado c3("Juan", 5);
    subdelegado c4;
    return 0;
}
```

14.2. Ejemplo 2.

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

class coordinador {
public:
    int curso = 0;
    string nombre = "indefinido";
    int current = 2025;

    coordinador() {
        // No imprimir nada aquí
    }

    coordinador(const string& name, int count) : nombre(name),
       curso(count) {
        // No imprimir aquí tampoco
    }

    virtual ~coordinador() {
        cout << "Elimina el coordinador" << endl;
    }

    // Método virtual para inicialización personalizada
    virtual void inicializar() {
        cout << "Este es el principal " << nombre << " de " <<
           curso << " curso" << endl;
    }

    virtual void contrato(int ye) {
        cout << "El fin de mi contrato es " << (ye + current)
           << endl;
    }
};

class subdelegado : public coordinador {
public:
    subdelegado(const string& name, int count) :
       coordinador(name, count) {}

    void inicializar() override {
        cout << "Este es el subdelegado " << nombre << " de "
           << curso << " curso" << endl;
    }

    ~subdelegado() {
        cout << "Elimina el subdelegado" << endl;
    }
```

```
};

int main() {
    coordinador c1;
    c1.inicializar();

    coordinador c2("Laura", 3);
    c2.inicializar();
    c2.contrato(10);

    subdelegado c3("Juan", 5);
    c3.inicializar();

    return 0;
}
```

Por consola:

```
Este es el principal indefinido de 0 curso
Este es el principal Laura de 3 curso
El fin de mi contrato es 2035
Este es el subdelegado Juan de 5 curso
Elimina el subdelegado
Elimina el coordinador
Elimina el coordinador
Elimina el coordinador
```

14.3. Ejemplo 3.

```
#include <iostream>
using namespace std;
struct A {
    A() { cout << "A()" << endl; }
    A(const A& a) { cout << "A_COPY" << endl; }
    A& operator = (const A& a) {
        cout << "A_EQUAL" << endl;
        return *this;
    }
};
//nueva clase:
struct B :A {
    B() {
        cout << "B()" << endl;
    }
    B(const B& b) { cout << "B_COPY" << endl; }
    B& operator + (const B& b) {
        cout << "B_SUM" << endl;
        return *this;
    }
};

struct C :A {
    C() { cout << "C()" << endl; }
    C(const C& c) { cout << "C_COPY" << endl; }
    C& operator = (const C& c) {
        cout << "C_EQUAL" << endl;
        return *this;
    }
};
//el main
int main() {
    B b1, b2(b1);
    b1 = b2;
    b1 + b2;
    C c1, c2(c1);
    c1 = c2;
    return 0;
}
```

Salida por consola:

```
A()
B()
A()
B_COPY
A_EQUAL
B_SUM
```

```
A()
C()
A()
C_COPY
C_EQUAL
```

14.4. Ejemplo 4. Junio 2025.

```
#include <iostream>
using namespace std;
struct A {
    A() { cout << "+A" << endl; }  // Constructor por defecto
    A(A& a) { cout << "COPY_A" << endl; }  // Constructor de
       copia (¡no es const!)
    A &operator=(const A& a) {  // Operador de asignación
        cout << "EQUAL_A" << endl;
        return *this;
    }
    ~A() { cout << "-A" << endl; }  // Destructor
};

struct B:A{
    virtual ~B(){cout<<"-B";}
};

struct C : B {
    C() { cout << "+C" << endl; }  // Constructor
    C(C &c) { cout << "COPY_C" << endl; }  // Constructor de
       copia
    C &operator=(const C& c) {  // Operador de asignación
        cout << "EQUAL_C" << endl;
        return *this;
    }
    ~C() { cout << "-C"; }
};
```

Para el siguiente `main`:

```
int main(){
    B b1;                 /*(1)*/
    C c1;                 /*(2)*/
    b1 = c1;              /*(3)*/
    c1 = c1;              /*(4)*/
    A* pa = new C(c1);    /*(5)*/
    delete(pa);           /*(6)*/
}                         /*(7)*/
```

Comentarios a realizar:

- Línea (1). Se declara un objeto `b1` de tipo `B`. Esto invoca primero el constructor por defecto de la clase base `A`, que imprime `+A`. La clase `B` no tiene constructor explícito, por lo que no imprime nada adicional en este paso.
- Línea (2). Se declara un objeto `c1` de tipo `C`. Esto invoca el constructor por defecto de `C`, que primero llama al constructor de `A` (imprimiendo `+A`) y luego imprime `+C` desde el cuerpo del constructor de `C`.

- Línea (3). Se realiza una asignación `b1 = c1`. Debido a que no hay un operador de asignación específico para `B = C`, se produce **object slicing** y se invoca el operador de asignación de la clase base `A`, el cual imprime `EQUAL_A`.
- Línea (4). Se realiza la asignación de un objeto `C` a sí mismo (`c1 = c1`). Esto invoca el operador de asignación definido en la clase `C`, que imprime `EQUAL_C`. Aunque se trata de una autoasignación, el operador no tiene protección contra ella.
- Línea (5). Se crea dinámicamente un nuevo objeto de tipo `C` usando el constructor de copia con `new C(c1)`. Como el constructor de copia de `C` no invoca explícitamente al constructor de copia de la clase base, se utiliza el constructor por defecto de `A` (imprimiendo `+A`). Luego se ejecuta el cuerpo del constructor de copia de `C`, imprimiendo `COPY_C`. El puntero resultante se guarda en `pa`, de tipo `A*`.
- Línea (6). Se libera la memoria del objeto apuntado por `pa` usando `delete`. Gracias a que el destructor de `B` es virtual, se llama correctamente al destructor de `C`, luego al de `B`, y finalmente al de `A`, imprimiendo `-C-B-A` en ese orden.
- Línea (7). Al finalizar `main()`, se destruyen los objetos automáticos en orden inverso al de su construcción. Primero se destruye `c1`, invocando los destructores de `C`, `B` y `A` (imprimiendo `-C-B-A`). Luego se destruye `b1`, invocando los destructores de `B` y `A` (imprimiendo `-B-A`).

14.5. Ejemplo 5.

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

class complex {
    float r = 0;
    float img = 0;
public:
    complex(){
        print();
    };

    complex(float a, float b){
        r = a;
        img = b;
        print();
    }
    ~complex() {
        cout << "Numero complejo destruido" << endl;
    }
    void print() {
        cout << "El numero complejo tiene la forma " << r << "
           " << img << "j" << endl;
    }

    complex& operator += (const complex& other) { //const!!
        r += other.r;
        img += other.img;
        return *this;
    }

    complex& operator -= (const complex& other) {
        r -= other.r;
        img -= other.img;
        return *this;
    }

    bool operator == (const complex& other) {
        return (((r == other.r) && (img == other.img)) ? 1 : 0);
    }
};

class plano{
    complex v1;
    complex v2;
public:
    plano() {
        cout << "Constructor por defecto de plano" << endl;
```

```
    }
    plano(const complex& a, const complex& b) : v1(a), v2(b) {
        cout << "Constructor con parametros de plano" << endl;
    }
};


int main() {
    complex C(1, 2);
    complex D(2, 3);
    D += C;
    D.print();
    if (D == C) {
        cout << "D es igual a C" << endl;
    } else {
        cout << "D es diferente de C" << endl;
    }
    plano P1(D,C);
    return 0;
}
```

Salida por consola:

```
El numero complejo tiene la forma 1 2j
El numero complejo tiene la forma 2 3j
El numero complejo tiene la forma 3 5j
D es diferente de C
Constructor con parametros de plano
Numero complejo destruido
Numero complejo destruido
Numero complejo destruido
Numero complejo destruido
```

14.6. Ejemplo 6. Sobrecarga opcional.

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

class complex {
    float r = 0;
    float img = 0;
public:
    complex(){
        print();
    };

    complex(float a, float b){
        r = a;
        img = b;
        print();
    }
    ~complex() {
        cout << "Numero complejo destruido" << endl;
    }
    void print() {
        cout << "El numero complejo tiene la forma " << r << "
           " << img << "j" << endl;
    }

    complex& operator += (const complex& other) { //const!!
        r += other.r;
        img += other.img;
        return *this;
    }

    complex& operator -= (const complex& other) {
        r -= other.r;
        img -= other.img;
        return *this;
    }

    friend bool operator==(const complex& a, const complex& b);
};

bool operator==(const complex& a, const complex& b) {
    return (a.r == b.r) && (a.img == b.img);
}

int main() {
    complex C(2, 3);
    complex D(2, 3);
    D += C;
```

```
    D.print();
    if (D == C) {
        cout << "D es igual a C" << endl;
    } else {
        cout << "D es diferente de C" << endl;
    }
    return 0;
}
```

Salida por consola:

```
El numero complejo tiene la forma 2 3j
El numero complejo tiene la forma 2 3j
El numero complejo tiene la forma 4 6j
D es diferente de C
Numero complejo destruido
Numero complejo destruido
```

14.7. Ejemplo 7. Sobrecarga de ostream.

```
#include <iostream>
using namespace std;
class A {
    int valor;
public:
    A() {
        cout << "+A" << endl;
    }
    A(int ini) {
        valor = ini;
        print();
    }
    void print() {
        cout << "El valor de A es " << valor << endl;
    }
};
class B : public A {
    int valor = 1;
public:
    B() {
        cout << "+B" << endl;
    }
    B(int ini) : A(ini) {
        cout << "El valor de B es " << valor << endl;
    }
    friend ostream& operator<< (ostream& co, const B& id);
};

inline ostream& operator<< (ostream& co, const B& id) {
    co << "B tiene " << id.valor << " bober kurwa";
    return co;
}

int main() {
    A a;
    B b(5);
    cout << b << endl;
    return 0;
}
```

Salida por consola:

```
+A
El valor de A es 5
El valor de B es 6
B tiene 6 bober kurwa
```

14.8. Laboratorio Junio 2025.

Enunciado:

```
#include <iostream>
#include <cmath>

using std::cout, std::endl, std::abs, std::ostream;

int mod(int a, int b) { // máximo común divisor por euclides
    return abs(b == 0 ? a : mod(b, a % b));
}

class Racional {
    int num{}, den{1}; // el denominador >=1 siempre
    void normaliza(); // simplifica num y den por mod
    public:
    Racional(int num = 0) : num{num} {} // inicialmente siempre
       es un entero
    Racional& operator+=(const Racional&);
    Racional& operator*=(const Racional&);
    Racional& operator/=(const Racional&);
    friend ostream& operator<<(ostream& os, const Racional&);
};

Racional& Racional::operator+=(const Racional& r) {
   //significado? porque Racionall::?
    num = num * r.den + r.num * den;
    den = den * r.den;
    normaliza();
    return *this;
}

Racional operator+(Racional a, const Racional& b) {
    return a += b;
}
```

```
//CÓDIGO A RELLENAR POR EL ALUMNO
ostream& operator<<(ostream& os, const Racional& a) {
    return os << "[" << a.num << "/" << a.den << "]";
}

void Racional::normaliza() {
    auto div = mcd(num, den);
    num /= div;
    den /= div;
    if (den < 0) {
        den = -den;
        num = -num;
    }
```

```
}

Racional operator*(Racional a, const Racional& b) {
    return a *= b;
}

Racional& Racional::operator*=(const Racional& r) {
    num = num * r.num;
    den = den * r.den;
    normaliza();
    return *this;
}

Racional operator/(Racional a, const Racional& b) {
    return a /= b;
    }

Racional& Racional::operator/=(const Racional& r) {
    num = num * r.den;
    den = den * r.num;
    normaliza();
    return *this;
}
```

14.9. Ejemplo 8.

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

class Tiempo {
    int hora;
    int minuto;
public:
    explicit Tiempo(int h = 0, int m = 0) {
        if (m < 0) m = 0;
        hora = h < 0 ? 0 : (h + m / 60);
        minuto = m % 60;
    }
    Tiempo operator+(const Tiempo& other) const {
        int totalMin = minuto + other.minuto;
        int totalHora = hora + other.hora + totalMin / 60;
        totalMin %= 60;
        return Tiempo(totalHora, totalMin);
    }
    bool operator<(const Tiempo& t2) {
        return((t2.hora > hora) && (t2.minuto > minuto));
    }
    friend ostream& operator<<(ostream& os, const Tiempo& t);
    Tiempo operator+(const int& minutos);
    bool operator==(const Tiempo& t1);
    bool operator>=(const Tiempo& t1);
};

inline ostream& operator<<(ostream& os, const Tiempo& t) {
    os << "El tiempo es " << t.hora << " horas y las " <<
       t.minuto << " minutos" << endl;
    return os;
}
Tiempo Tiempo::operator+(const int& minutos) {
    return Tiempo(hora, minuto + minutos);
}
inline bool Tiempo::operator==(const Tiempo& t1) {
    return (hora == t1.hora) && (minuto == t1.minuto);
}

inline bool Tiempo::operator>=(const Tiempo& t1) {
    return (hora > t1.hora) && (minuto > t1.minuto);
}
int main() {
    Tiempo t1(2, 40);
    Tiempo resultado = t1 + 14;
    cout << t1;
    cout << resultado;
```

```
    return 0;
}
```

Salida por consola:

```
El tiempo es 2 horas y las 40 minutos
El tiempo es 2 horas y las 54 minutos
```

14.10. Ejemplo 9.

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

template <class T>
class B {
public:
    int f(int i) {
        return derived()->f_imp(i);
    }
    T* derived() {
        return static_cast<T*>(this);
    }

    int f_impl(int i) { return i; }
    virtual ~B() { cout << "-B" << endl; }

    protected:
    int i_ = 0;
};


class C1 : public B<C1> {
public:
    C1() { cout << "C1+" << endl; }
    int f_imp(int i) { i_ += i; return i_; }
    ~C1() { cout << "-C1" << endl; }
};

class C2 : public B<C2> {
public:
    int f_imp(int i) { i_ += 2 * i; return i_; }
};


template<class T>
int foo(int i, B<T>& d) { return d.f(i); }

int main() {
    {
        C1 c1;                          // (1) Constructor de C1
           -> imprime "C1+"
        cout << foo(10, c1) << endl;    // (2) llama a c1.f(10) →
           C1::f_imp(10) suma 10 a i_ → imprime 10
    }                                   // (3) Destructor de C1 →
       imprime "-C1"
    // (4) Destructor de B<C1> → imprime "-B"
    {
```

```
        C2 c2;
        cout << foo(10, c2) << endl;   // (5) llama a c2.f(10) →
           C2::f_imp(10) suma 20 a i_ → imprime 20
    }                                  // (6) Destructor de
       B<C2> → imprime "-B"
}
```