VARIABLES Y TIPOS DE DATOS DE UN PROGRAMA EN C++

Las variables en los programas nos permiten almacenar valores. Dependiendo el tipo de valor que queremos guardar es la cantidad de memoria que se reserva (memoria física). Debido a lo anterior se definen diferentes tipos de datos de manera que se sabe exactamente cuánta memoria es la que se va a utilizar para guardar determinada información. A continuación los tipos de datos de c++ y la cantidad de memoria utilizada para almacenar dicha información:

Nombre	Descripción	Tamaño en memoria
Int	Números enteros	4 bytes
Float	Punto flotante de precisión simple	4 bytes
Double	Punto flotante de doble precisión	8 bytes
Char	Caracter	1 byte
Bool	Booleano (verdadero / falso)	
Short int	Entero corto	2 bytes
Long int	Entero largo /muy largo	8 bytes

Operadores

Los operadores permiten "operar" sobre los valores de las variables. Hay diferentes tipos de operadores. A continuación, una lista de los más usados (hay más):

Operador	Descripción
+,-,*,/, %	Operadores aritméticos
>, <, >=, <=, !=, ==	Operadores relacionales (comparación)
+=, -=, *=, /=,%=	Asignación compuesta
++, --	Incremento y decremento
&&, ||	Operadores lógicos
&, |, ^, ~, <<, >> (and, or, xor, not, left shift, right shift)	Bitwise operators
<<, >> (stream insertion, stream extractor)	Stream Operator (overloaded bitwise)
::	Scope resolution
new, delete	Memory allocation / deallocation
&, *	Reference, dereference

Estructura general de un programa en C++

De manera general podemos dar una estructura a nuestro código en c++ y separarlo en diferentes secciones

- Directivas del Preprocesador
  - Librerías
  - Inicializaciones del preprocesador
- Definiciones globales
- Funciones / Clases
- Función principal (código principal)

A continuación se muestra la estructura general de un programa en C++

```
#include<iostream>
#include<stdlib.h>
#include"arithmetic.h"
#include"wolfram_alpha.h"
```

Directivas del preprocesador
Librerías

```
#define PI = 3.1416;
```

Directivas del preprocesador
Definición de constantes

```
int resuelveSituacionProblema(int a, double b)
{
    int resultado = 0;
    return resultado;
}

bool noMePongoLasPilasEnElCurso(bool melaspongo)
{
    bool repruebo = true;
    bool apruebo = false;
    if (melaspongo == false)
        return repruebo;
    else
        return apruebo;
}
```

Funciones

Recuerda que las funciones también pueden estar en archivos separados (hpp y cpp). **¿Cuál crees que sea la ventaja de que estén separados en archivos separados?**

```
int main()
{
    resuelveSituacionProblema(10, 8.8);
    noMePongoLasPilasEnElCurso(false);
}
```

Función principal / Código principal

Sección de LIBRERÍAS

Permite incluir funciones de código a partir de otros archivos (reusar código). Para incluir una librería se utiliza la palabra reservada "**include**" y el símbolo "**#**". Observa que el nombre de la librería se puede incluir

- Entre comillas. Significa que la librería se busca en el **directorio actual** en el que está el código de programa
- Entre signos de mayor qué y menor qué. La librería se busca en el **directorio del sistema**

A continuación se muestra un ejemplo de cómo incluir librerías que están en el sistema o librerías propias.

```
#include<iostream>
#include<stdlib.h>
#include"arithmetic.h"
#include"wolfram_alpha.h"

using namespace std;
```

Además, es IMPORTANTE QUE NOTES que en el código aparece `using namespace std;` que indica que estoy utilizando el espacio de nombres **std** que hace referencia a la librería estándar.

Namespaces. Muchas de las funciones más comunes o más usadas están agrupadas en lo que se llama la "librería estándar" (**std**). Esta agrupación de nombre "std" se llama "espacio de nombres" (**namespaces**). "std" es el nombre que alguien le dio al **namespace**, pero podría ser cualquier nombre.

Ejemplo. Agrupar las funciones en **namespaces** es como entrar a la biblioteca del campus e ir directamente a la sección de "computación". Dividir la biblioteca en secciones permite buscar fácilmente libros.

Dividir nuestro código en **namespaces** permite organizar mejor mis funciones y me facilita buscar las funciones que otros programadores han hecho (o yo mismo). Ej. Imagina que defines tus propias funciones aritméticas y la guardas en tu propia librería y espacio de nombres y lo llamas "arithmetic", pero luego quieres incluir las funciones matemáticas de "Wolfram Alpha". (a continuación, se muestran las funciones incluidas en cada **namespace**)

Si incluyes ambas librerías en tu código, entonces ¿cómo sabe el programa cuál de las dos funciones "**add**" usar?

```
#include "arithmetic.h"
#include "wolfram_alpha.h"

int main()
{
    add(5,5);
}
```

¿Cómo sabe c++ cuál **add** usar? ¿usa el de arithmetic o el de wolfram?

La respuesta es simple: hay que indicar a qué **namespace** pertenece la función que voy a usar. Esto se puede hacer de 2 formas:

1. Utilizando las palabra reservadas **using namespace**

```
#include "arithmetic.h"
#include "wolfram_alpha.h"

using namespace arithmetic;
int main()
{
    add(5,5);
    wolfram_alpha::add(8,8);
}
```

2. A través del operador de desambiguación (the **scope resolution operator is** **::**)

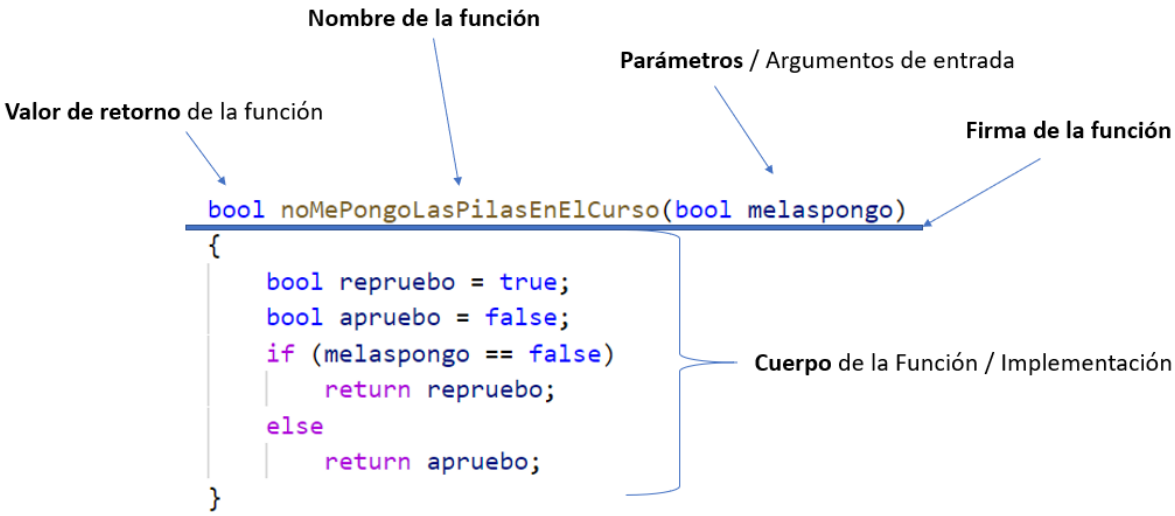
```
#include "arithmetic.h"
#include "wolfram_alpha.h"

int main()
{
    arithmetic::add(5,5);
    wolfram_alpha::add(8,8);
}
```

Sección de Directivas de Preprocesador

Esta sección permite mandar comandos de procesamiento, es decir, todo aquello que quiero que se realice el compilador de C++ antes de iniciar el proceso de compilación de mi código. Nota: cuando lleguemos a programación Orientada a Objetos utilizaremos la directiva del preprocesador **#pragma once**, así que de momento no daremos más detalles.

SECCIÓN FUNCIONES / CLASES

En esta sección se definen las funciones o clases que usaremos en nuestro programa. Es importante que separemos nuestras funciones / clases en diferentes archivos porque eso facilita reusar nuestro código (compartirlo o usarlo en otro programa). Imagina qué pasaría si pones 100 funciones matemáticas y 100 funciones para manipular archivos en un mismo archivo, entonces, cuando quieras ocupar el subconjunto de las funciones aritméticas deberás importar las 200 funciones y no únicamente el subgrupo de las aritméticas (overhead). Es muy importante que identifiques las partes de una función. A continuación se detallan:



Partes de una función

*** Firma de la función.** Está formada diferentes elementos:

- **Nombre** de la función. Identificador único (más adelante veremos cómo nombrar dos funciones con el mismo nombre).
- **Valor de retorno**. Permite indicar qué valor devuelve la función (**int**, **void**, etc.) Recuerda que **void** significa que la función no regresa ningún valor.
- **Parámetros** entrada de la función. Son los valores de entrada que vas a procesar en el cuerpo de tu función. Sí, debes estar pensando en ¿cuál es a diferencia con una variable? La diferencia es el lugar donde se declaran. Si está declarada dentro del cuerpo de una función, entonces se llama variable. Si está declarada en la firma de una función, entonces se llama parámetro.

*** Cuerpo de la función / Implementación. El cuerpo de la función es donde están las** instrucciones a ejecutar por nuestro programa. Recuerda que el cuerpo de una función puede estar compuesto por declaración de variables, uso de estructuras condicionales (if), uso de estructuras de repetición (for), diferentes instrucciones (cout, a+b, etc.). Además es importante que recuerdes que el cuerpo/implementación de una función está delimitado por llaves/brackets/braces {}. Estas mismas llaves son las que permiten conocer el **scope** de una función. Recuerda que la variable de una función (delimitada por *braces*) sólo es visible para esa función y no para otras. Esa visibilidad está delimitada por el scope y el scope está delimitado por las llaves {}

FUNCIÓN MAIN

Es la función principal del programa. A partir de ella inicia el flujo de cualquier programa. Sus principales características son:

- Debe devolver un entero. Usualmente se asocia que devolver 0 es un estado “correcto” o que el programa está finalizando de manera adecuada
- Puede o no tener parámetros. Los parámetros se le agregan cuando deseamos que el programa lea valores desde la terminal en la misma línea que se ejecuta el programa.

```
'classPreparation $ ./suma 1 2
```

Ej. La siguiente línea ejecuta el programa “suma” y le pasa como parámetros de entrada 2 valores enteros

Nota: Las funciones las volveremos a abordar más adelante como parte del contenido del curso

Hora que ya conoces la estructura general de un programa en C++ repasaremos breve repaso de los conceptos fundamentales de programación (condicionales y estructuras de control).

Condicionales

En un programa/código es muy importante la toma de decisiones. Es decir, que nuestro código pueda “decidir” ejecutar un bloque de instrucciones si cierta condición se cumple y si no, entonces realizar un segundo bloque de instrucciones. Para lograrlo se cuenta con la instrucción **IF**. A continuación se muestra la estructura de un condicional:

```
if (age < 18)
{
    cout << "eres mayor de edad" << endl;
}
else
{
    cout << "eres menor de edad" << endl;
}
```

Los condicionales tienen 4 elementos

1. Palabra reservada **if**
2. Condición a evaluar (recuerda que una condición es una expresión que puede ser evaluada como verdadero o falso). Nota: Las condiciones siempre se deben escribir entre paréntesis.
3. Cuerpo de la condición **if** (bloque de instrucciones que se ejecutan si la condición se es verdadera)
4. Palabra reservada **else** que indica qué debe de hacer el código en caso de que la condición no se cumpla
5. Cuerpo del bloque **else** (bloque de instrucciones que se ejecutan si la condición no se cumple

El programa anterior no considera aquellos casos en los que un usuario coloca un valor negativo de la variable “age” por lo que a continuación tenemos el mismo ejemplo, pero con una condición compuesta que no considera cuando un usuario pone un valor negativo de la edad

```
if ((age >=0 ) && (age < 18))
{
    cout << "eres menor de edad" << endl;
}
else
{
    if(age >= 18)
    {
        cout << "eres mayor de edad" << endl;
    }
}
```

LOOPs

Otra estructura importante de un programa son los ciclos (LOOPs). Sirven para poder repetir bloques de operaciones. Hay tres formas de hacer LOOPs en C++

1. For
2. While
3. Do while

Ciclos For

Los ciclos FOR están compuestos por los siguientes elementos

- Inicialización (variable contador inicializada)
- Condición de paro (sentencia que puede ser evaluada como verdadero o falso)
- Incremento (cómo debe crecer la variable contador -generalmente se usan los operadores de incremento)

A continuación un ejemplo. Ej. Ciclo que suma todos los números que hay entre cero y 20

```
int sum = 0;
for (int i=0; i<20; i++)
{
    sum+=i;
}
cout << "la suma de los 20 números es " << sum;
```

Ciclos While

En los ciclos while no se define cuántas veces se va a repetir un ciclo. El ciclo se repite mientras que se cumpla una determinada condición. A continuación un ejemplo: Ej. Ciclo que suma números consecutivos y que se detiene cuando la suma es igual a 19900

```
int consecutivo = 0;
int sum = 0;
while (sum <= 19900)
{
    cout << " + " << consecutivo++;
    sum+=consecutivo;
}
cout << endl << "se sumaron " << --consecutivo << "y el resultado es " << sum;
```

Observa en el código anterior que el programa no sabe cuántas sumas va a realizar, pero sigue sumando hasta que se cumple la condición.

Ciclos Do While

Este tipo de loops es muy similar al while, pero la diferencia es que el bloque de código a repetir SIEMPRE SE EJECUTA AL MENOS UNA VEZ sin importar si la condición se cumple o no (a diferencia con el WHILE que sólo se ejecuta si la condición es verdadera). A continuación un ejemplo:

```
do
{
    cout << "este código se va a ejecutar al menos una vez" << endl;
    cout << "después de la ejecución del bloque de insctrucciones se evalúa la condición " << endl;
}while(0 < -100);
```

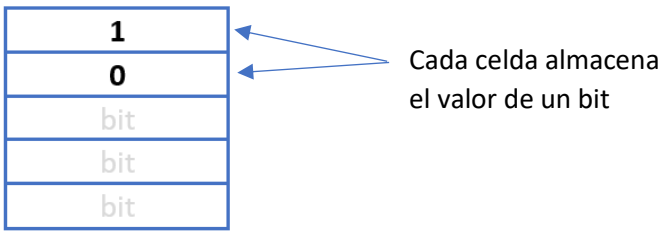
MI PROGRAMA EN C++ Y LA MEMORIA DE MI COMPUTADORA.

Ahora ya tienes más clara la estructura básica de un programa en C++, pero No te has preguntado **¿cómo se guarda mi programa en la memoria de mi computadora? ¿Cómo se guardan la variables?**

Lo primero es recordar es tu PC lo "único" que entiende es código binario. Recuerda que la memoria está compuesta por transistores y capacitores.

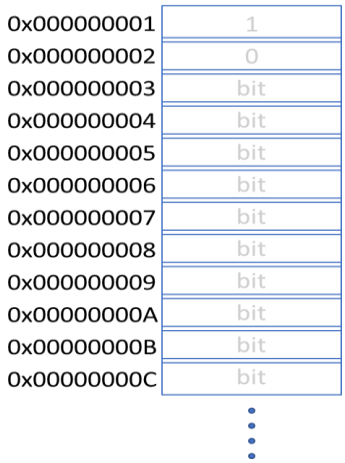
- Capacitor. El capacitor sólo guarda un estado (está prendido o apagado)
- El transistor. Es un switch que lee el estado del capacitor o que cambia su estado (lo prende o lo apaga)

Lo anterior es lo que sucede a nivel de hardware, pero para explicarlo mejor y con fines educativos asumiremos de manera muy básica que la memoria de la computadora es como una lista de valores (Lista si pensamos en Python) o como un Arreglo (Arreglo si pensamos en C++ -después daremos más detalles-), donde el valor de cada celda corresponde a un bit (prendido 1, apagado 0). Observa en el siguiente diagrama cómo cada celda almacena el valor de un bit.

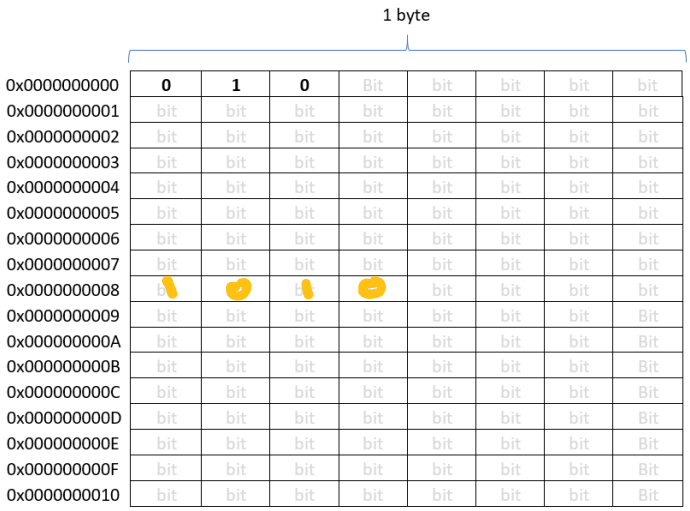


Pero, el diagrama anterior no te hace pensar ¿cómo sé en qué celda hay un cero y en qué celda hay un uno? La respuesta es muy sencilla: hay que colocar un índice o un valor que esté asociado a cada celda. Ej. Piensa en algo tan simple como tu casa. ¿Cómo sabe el cartero dónde entregar tus cartas? Por tu dirección que tiene **un número o índice**.

Dado que la memoria sólo entiende binario, entonces lo índices se representan con números binarios. Sin embargo, eso sería muy complejo de leer e interpretar para el ser humano por lo que se buscó un sistema numérico que fuera más amigable con el ser humano (se seleccionó el sistema hexadecimal). Si buscaban algo más amigable ¿por qué no eligieron el sistema decimal para representar el índice de cada celda? Simple, porque es más fácil convertir números binarios en números hexadecimales y viceversa. Así, llegamos a la siguiente representación en la que la dirección de memoria de cada celda está representada por números binarios.



Todo va bien, pero para facilitar aún más la “interpretación” de la memoria se decidió agrupar los bits en bloques de 8 bits y se les llamó Bytes. Así, pasamos de tener una lista a tener una matriz de bits como se ve en la siguiente figura



Bueno, todo está muy bonito hasta aquí, pero ¿en mi programa cómo veo lo anterior?

Para poder verlo necesitamos una forma de obtener la dirección de una variable en mi programa. Para ello utilizaremos un nuevo **operador**:

- **&** Operador "**address of**"

El operador "address of" nos dice la dirección en memoria en la que está almacenada una variable (también tiene otros usos). Veámoslo con el siguiente código:

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

int main()
{
    int a = 5;
    cout << &a;
}
```

El operador "address of" se coloca antes de la variable.

Esta línea se lee como "imprime la dirección de A"

El resultado de compilar y ejecutar el código anterior es:

```
ariellucien@L0:
0x7fffe66292f4
```

Que es la dirección de memoria en la que se encuentra almacenado el valor 5. Agreguemos más variables a nuestro código

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

int main()
{
    int a = 5;
    int b = 10;
    int c = 15;
    cout << "Dirección de a es" << &a << endl;
    cout << "Dirección de b es" << &b << endl;
    cout << "Dirección de c es" << &c << endl;
}
```

El resultado de compilar y ejecutar el código anterior se muestra a continuación:

```
Dirección de a es0x7ffff9e832ac
Dirección de b es0x7ffff9e832b0
Dirección de c es0x7ffff9e832b4
```

Realicemos un pequeño análisis del resultado anterior. Tomemos los últimos valores del resultado:

```
Dirección de a es0x7ffff9e832ac
Dirección de b es0x7ffff9e832b0
Dirección de c es0x7ffff9e832b4
```

Ahora hagamos una conversión de cada número a binario:

Hexadecimal	Binario
ac	172
b0	176
b4	180

¿Qué puedes deducir a partir de los resultados anteriores?

Sí. Puedes deducir el tamaño de cada tipo de datos. Con lo anterior sabemos que en mi computadora en particular un **entero** se guarda en **4 bytes**

Ejercicio a.

Crea un programa que te permite resolver las siguientes preguntas

1. ¿Cuántos bytes se utilizan para almacenar los siguientes tipos de datos?

a. Entero (**int**)
b. Entero corto (**short**)
c. Carácter (**char**)
d. Punto flotante (**float**)
e. Doble precisión (**double**)
f. Boolean (**bool**)
g. Long int (**long int**)
h. Long double (**long double**)

Tipos de datos de C++

Conociendo más de la memoria de mi PC y de C++

Ahora conoces un poco mejor cómo se almacena la información en la memoria. Y ¿qué tanta diferencia hay entre c++ y Python? Resulta que pyhton nos hace la vida muuuuy fácil para escribir programas, pero por dentro hace exactamente lo mismo que c++ (hay direcciones de memoria, tipos de datos para guardar diferentes tipos de valores), entonces ¿cuál crees que es la ventaja de usar C++ en lugar de Python? Sí, la ventaja más clara es que nuestros programas en c++ son más óptimos (son más rápidos y pueden usar menos recursos). Entonces un programa en c++ es más complejo de escribir, pero más veloz y con menos recursos computacionales.

Bueno, sigamos con la explicación de cómo C++ administra la memoria. Volvamos a nuestro programa original:

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

int main()
{
    int a = 5;
    int b = 10;
    int c = 15;

}
```

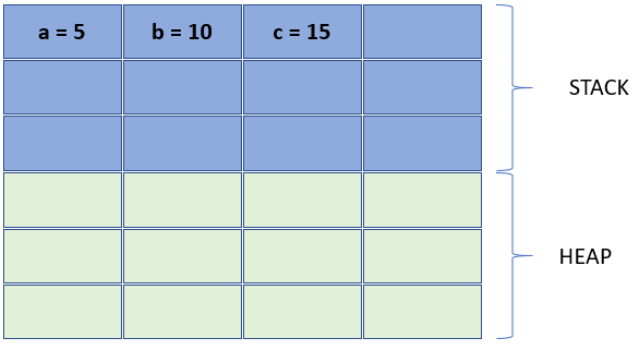
Dado el programa anterior ¿Cuánta memoria se reserva para su ejecución? Sí, sólo 12 bytes. Esa memoria se reserva al inicio de la ejecución del programa porque el compilador sabe que sólo utilizará 12 bytes, es decir, ese programa NO puede utilizar más de 12 bytes porque cuando se compiló se determinó que sólo utilizaría 12 bytes. Esos 12 bytes son fijo, o no se mueven. En términos de programación se dice que esos 12 bytes son estáticos y el área de memoria en el que se reservan se llama “pila” o “**STACK**”. Al ser estática dicha memoria, entonces hay dificultades.

Ej. Imagina que tienes un programa para almacenar a todos los alumnos de tu salón (30 alumnos) y que para cada alumno vas a utilizar 100 bytes, entonces, cuando se compila el programa se indica que en total el programa va a utilizar 30,000 bytes. Esos 30 kbytes (kilo bytes) se reservan en el stack (espacio estático de la memoria), pero ¿qué pasaría si llegaran más alumnos? ¿podrías almacenarlos en tu programa? No, no podrías porque al compilar el programa se determinó que era necesario sólo reservar 30,000 bytes. Entonces ¿cómo hace c++ para permitir que los programas puedan utilizar la memoria libremente, que no sea estática a reserva?

Para poder permitir la administración dinámica de la memoria c++ provee de **apuntadores** y un espacio de memoria que puede ser administrado libremente y es llamado **HEAP**. Veamos el siguiente diagrama que muestra parcialmente cómo está distribuida la memoria:

Regresemos a nuestro programa y veamos cómo se almacena en memoria

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

int main()
{
    int a = 5;
    int b = 10;
    int c = 15;

}
```



Entonces cómo se podemos administrar la memoria de manera dinámica: **APUNTADORES**

APUNTADORES

Los apuntadores permiten administrar la memoria dinámica de un programa. Se llaman apuntadores porque **apuntan a una dirección de memoria.** Los **apuntadores** son variables "especiales" que no guardan valores nativos (int, double,etc.) sino que **guardan direcciones de memoria** (por eso dedicamos toda la primera parte en hablar de direcciones de memoria).

NOTA: Es muy común que cuando se habla de apuntadores se hable de REFERENCIAS. "El apuntador guarda **REFERENCIAS"**. Hablar de referencias es lo mismo que hablar de direcciones de memoria

A continuación aprenderemos cómo manipular apuntadores

- Declaración de apuntadores
- Asignación de valores a un apuntador
- De-Referenciación de un apuntador (acceso al contenido apuntado)
- Reservar memoria dinámica con un apuntador
- Liberar la memoria dinámica reservada con un apuntador

Declaración de apuntadores

Los apuntadores se declaran igual que las variables (recuerda que cuando declaras una variable es forzoso especificar qué tipo de datos va a almacenar). Se especifica el tipo de datos de la dirección que almacenan. Se declaran usando *

A continuación algunos ejemplos de declaración de apuntadores

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

int main()
{
    int *pointer2int;           // La variable pointer2int es un apuntador a una dirección que guarda un entero
    double *pointer2double;     // La variable pointer2double apunta a una dirección que guarda un valor double
    char *pointer2char;         // La variable pointer2char apunta a una dirección que guarda un caracter
}
```

Como verás la declaración de apuntadores es muy simple (sólo es necesario anteponer un * al nombre de la variable)

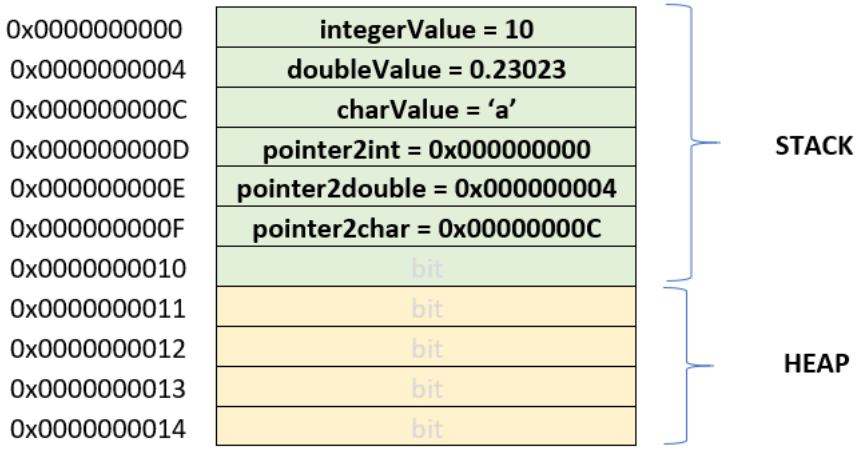
Asignar valores a un apuntador

Antes de asignar valores a un apuntador debes recordar que los apuntadores **ALMACENAN DIRECCIONES DE MEMORIA** ¿recuerdas cuál es el operador que obtiene la dirección de memoria de una variable? Sí, el ampersand **&**

Vemos algunos ejemplos de cómo asignar valores a un apuntador

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

int main()
{
    int integerValue = 10;
    double doubleValue = 0.23023;
    char charValue = 'a';

    // La variable pointer2int guarda la dirección de la variable integerValue
    int *pointer2int = &integerValue;

    // La variable pointer2double apunta a la dirección la variable doubleValue
    double *pointer2double = &doubleValue;

    // La variable pointer2char apunta a la dirección de la variable charValue
    char *pointer2char = &charValue;
}
```

Veamos un pequeño diagrama que representa lo que sucedió en memoria con el programa anterior



Observa cómo los **APUNTADORES SON VARIABLES QUE GUARDAN REFERENCIAS** de memoria. Seguro te estarás preguntando ¿no quedamos que los apuntadores permiten guardar valores dinámicos? ¿por qué están en el stack si el stack es el espacio estático? Tranqui, más adelante llegaremos a ese punto.

Es importante que sepas que, aunque los apuntadores guardan direcciones de memoria, NO SE PUEDE ASIGNAR directamente valores HEXADECIMALES. En el siguiente código se muestra lo que NO ES VÁLIDO HACER

```
int main()
{
    int*a = 0xff;
}
```

De-Referenciación de un apuntador (acceso al contenido de la dirección del apuntador)

Recuerda que dijimos que hablar de REFERENCIAS es lo mismo que hablar de direcciones de memoria. En pasos previos aprendiste cómo asignar direcciones de memoria a un apuntador, pero en nuestros programas nunca usaremos impresas direcciones de memoria, sólo vemos valores, entonces ¿cómo puedo acceder al valor al que apunta el apuntador? Es decir ¿cómo puedo acceder al contenido apuntado? Muy sencillo.

Acceder al contenido de un apuntador se llama De-Referenciar el apuntador. Se llama de-referenciar por que le voy a quitar la referencia, es decir la dirección, para quedarme solo con el valor.

Para de-referenciar se utiliza el operador *, queeeeeeé? WT..? el mismo asterisco es para declarara apuntadores y para de-referenciar? Sí, es el mismo. A continuación algunos ejemplos

```
void main3()
{
    int integerValue = 10;
    double doubleValue = 0.23023;
    char charValue = 'a';

    int *pointer2int = &integerValue;
    double *pointer2double = &doubleValue;
    char *pointer2char = &charValue;

    cout <<" la dirección que guarda pointer2int es " << pointer2int << " y su contenido es " << *pointer2int << endl;
    cout <<" la dirección que guarda pointer2double es " << pointer2double << " y su contenido es " << *pointer2double << endl;
    cout <<" la dirección que guarda pointer2char es " << pointer2char << " y su contenido es " << *pointer2char << endl;
}
```

Pointer o Dirección de memoria

Contenido o Dereferencia

Usos del **operador ***

- Sirve para declarar apuntadores
- Permite dereferenciar apuntadores (acceder a su contenido)

Operadores de De-Referenciación

Existen dos operadores que permiten dereferenciar apuntadores. Se pueden utilizar de manera indistinta (depende del programador cuál quiera usar)

- * El asterisco
- [ ] Los corchetes

A continuación un ejemplo

```cpp
int main()
{
    int integerValue = 10;
    double doubleValue = 0.23023;
    char charValue = 'a';

    int *pointer2int = &integerValue;
    double *pointer2double = &doubleValue;
    char *pointer2char = &charValue;

    cout <<" pointer2int dereferenciado con * " << *pointer2int << " y con [] " << pointer2int[0] << endl;
    cout <<" pointer2int dereferenciado con * " << *pointer2double << " y con [] " << pointer2double[0] << endl;
    cout <<" pointer2int dereferenciado con * " << *pointer2char << " y con [] " << pointer2char[0] << endl;
}
```

Dereferencia con *

Dereferencia con []

¡Y ahora la pregunta del millón! ¿Por qué crees que cuando se usan los corchetes [] para dereferenciar se coloca el índice cero? ¿qué te recuerda esto? ¿Qué puedes deducir a partir de esto? Envía tu respuesta directamente al chat de tu mentor 😊 (no tiene fecha, pero sé que te ayudará)

Con esto en mente ¿qué pasaría si cambio el valor al que apunta uno de mis apuntadores? ¿qué pasaría con valor de la variable original? Vemos el código y obtén tus conclusiones

```cpp
int main()
{
    int integerValue = 10;
    int *pointer2int = &integerValue;

    //ahora cambiemos al valor al que apunta pointer2int -dupliquémoslo-
    *pointer2int = *pointer2int*2;

    //vamos a imprimir el valo original
    cout << "el valor de la variable original integerValue ahora es " << integerValue;
}
```

Reservar Memoria Dinámica con Apuntadores

Hasta ahora sólo hemos reservado memoria en la parte estática (aún utilizando apuntadores), pero cómo podemos realmente explotar las ventajas de administrar “libremente” la memoria en mis programas?

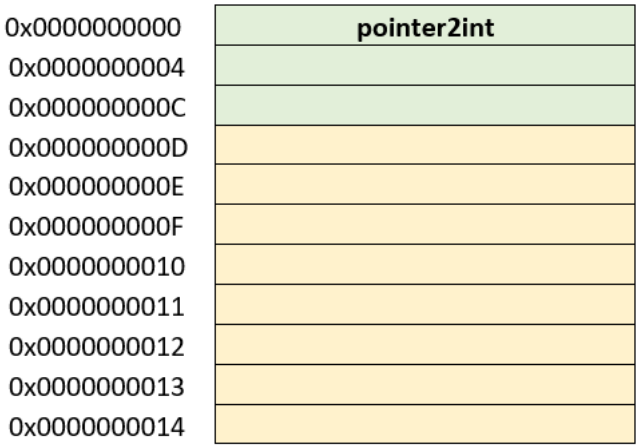
Para reservar memoria de manera dinámica hay 2 operadores:

- **malloc** esta es la forma de reservar memoria en C
- **new** esta es la forma de reservar memoria en C++ (en el interior de new, en las tripas, se utiliza malloc, pero para facilitar todo en C++ se creo una función más simple)

A continuación se muestra cómo se reserva memoria de forma dinámica

```
int main()
{
    int *pointer2int;

    pointer2int = new int;

    return 0;
}
```

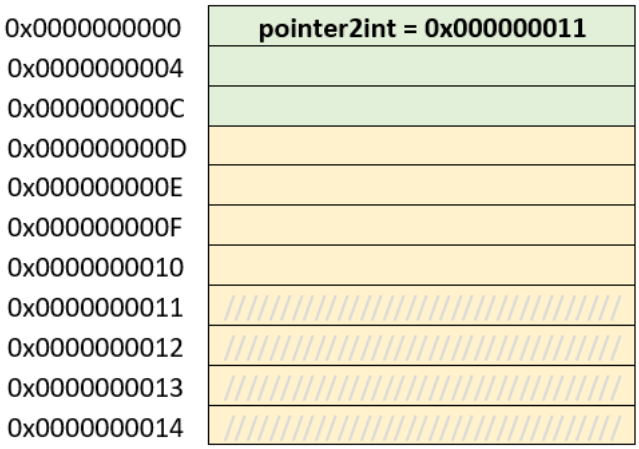
Donde `int *pointer2int;` es la declaración del apuntador. Luego, en lugar de asignarle una dirección de memoria (como en capítulos anteriores), le indicamos al compilador que reserve **nuevo** espacio de memoria y, siguiendo las reglas para las variables, es necesario indicar cuánto espacio es que se quiere reservar (espacio para un entero) `pointer2int = new int;` Ahora veamos cómo se ve el código anterior en memoria




```
int *pointer2int;
```

Cuando se ejecuta la instrucción anterior lo único que sucede en memoria es que se reserva espacio en el **stack** para una variable que es un apuntador. Observa que NO tiene valor o no apunta a nada

Luego cuando ejecutamos la siguiente instrucción:




```
pointer2int = new int;
```

Cuando ejecutamos la instrucción anterior se reserva espacio para almacenar un entero -4 bytes- Dicho espacio se reserva en el **heap**

Observa cómo el espacio se reservó al final de la memoria. Se utilizaron //// para indicar que es espacio reservado, pero nota que ese espacio NO TIENE NOMBRE.

¡Pregunta del segundo millón! Explica ¿por qué crees que se reservo la memoria del Heap al revés?

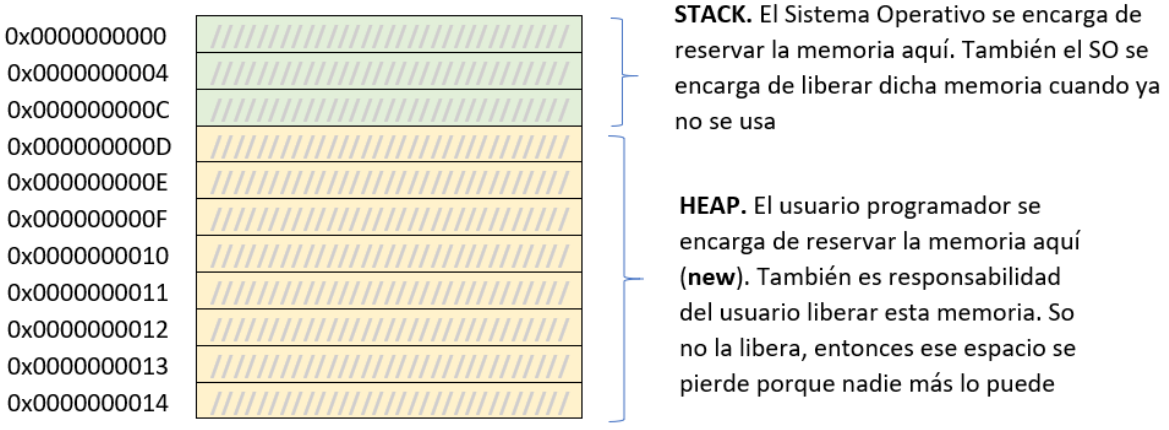
Envía tu respuesta directamente al whatsapp de tu mentor (a ver qué cara pone cuando le llenemos el whats de mensajes 😊)

A continuación más ejemplos de reserva de memoria dinámica. Además de la reserva asignaremos valores a los apuntadores.

```
int main()
{
    int *pointer2int;
    float *pointer2float;
    string *pointer2string;

    pointer2int = new int;
    pointer2float = new float;
    pointer2string = new string;

    *pointer2int = 323;
    *pointer2float = 323.3f;
    *pointer2string = "échale ganitas para no reprobar";
    return 0;
}
```

NOTA MUY MUY IMPORTANTE RESPECTO A LA MEMORIA

Es muy importante que sepas que TODO LO QUE SE RESERVA EN ESPACIO ESTÁTICO (MEMORIA EN EL STACK) es administrado por el sistema operativo. Esto es una gran ventaja porque cuando compilas tu programa, entonces EL COMPILADOR PIDE AL SISTEMA OPERATIVO QUE RESERVE ESPACIO EN LA MEMORIA y así lo hace, pero cuando termina tu programa ¿quién se encarga de liberar dicha memoria? Correcto, el sistema operativo es quien se encarga de liberar la memoria, porque él la reservó. ¿Qué pasa cuando tú reservas espacio de manera dinámica? ¿Quién se encarga de liberar la memoria dinámica reservada? Correcto, es TÚ RESPONSABILIDAD COMO PROGRAMADOR liberar la memoria que hayas reservado de manera dinámica!!!!!!



Liberar la memoria dinámica reservada con un apuntador

Para liberar la memoria dinámica se pueden utilizar 2 operadores

- **free** Es el operador de C que permite liberar memoria
- **delete** Es el operador de C++ para liberar memoria dinámica (muy en su interior delete funciona usando el free de c++)

se recomienda liberar la memoria cuando sabes que ya no vas a utilizar una variable. A continuación algunos ejemplos de cómo reservar y liberar memoria:

```
int main()
{
    int *pointer2int;
    float *pointer2float;
    string *pointer2string;

    pointer2int = new int;
    pointer2float = new float;
    pointer2string = new string;

    *pointer2int = 323;
    *pointer2float = 323.3f;
    *pointer2string = "échale ganitas para no reprobar";

    delete pointer2int;
    delete pointer2float;
    delete pointer2string;

    return 0;
}
```

Se debe liberar la memoria dinámica reservada antes de que termine nuestro programa

Funciones

Ahora vamos a hacer un pequeño repaso de funcione. Recuerda que las funciones son bloques de código cuyo principal objetivo es reutilizar código: “sí hago un programa que resuelve integrales, entonces cada vez que quiera resolver una integral sólo mando llamar mi programa/función sin tener que escribir toooodo el código necesario para resolver una integral. Los temas que vamos a tratar son (recuerda que los parámetros son las variables de entrada de una función):

- Pasar parámetros por valor
- Pasar parámetros por referencia
- Pasar parámetros por dirección

Pero, como estoy obsesionado con el la memoria de la computadora, entonces veamos qué sucede en memoria cuando definimos y usamos funciones (veremos un programa con funciones y paso a paso veremos qué sucede en memoria)

```
 87   int add(int num1, int num2)
 88   {
 89       return num1+num2;
 90   }
 91
 92   int multiply(int num1, int num2)
 93   {
 94       return num1*num2;
 95   }
 96
 97   int main2()
 98   {
 99       int num1 = 10;
100       int num2 = 20;
101
102       int res = add(num1, num2);
103   }
104
```

Sigamos paso a paso cada una de las instrucciones y vemos qué va sucediendo en memoria. Todo inicia en la línea 99

Primero se crean 3 variables en memoria

```
int num1 = 10;
int num2 = 20;

int res =
```

0x0000000000	**num1 = 10**
0x0000000004	**num2 = 20**
0x000000000C	**res =**
0x000000000D	
0x000000000E	
0x000000000F	
0x0000000010	
0x0000000011	
0x0000000012	
0x0000000013	
0x0000000014	

Luego viene la llamada a la función (aquí lo interesante). Si ves la firma de la función verás que tiene 2 parámetros de tipo entero. Esos parámetros se deben crear también en memoria

```
add(num1, num2);
```

0x0000000000	**num1 = 10**
0x0000000004	**num2 = 20**
0x000000000C	**res =**
0x000000000D	**num1 = 10**
0x000000000E	**num2 = 20**
0x000000000F	
0x0000000010	
0x0000000011	
0x0000000012	
0x0000000013	
0x0000000014	

¡¡¡¡Qué diablos!!!!!

¿Cómo puede haber en memoria dos variables que se llaman igual?

No entraré en detalle a lo que ocurre dentro de la función **add,** pero sí a lo que sucede cuando termina la función add y se devuelve el resultado de la operación

```
int res = add(num1, num2);
```

Observa cómo cuando se termina de ejecutar la instrucción **desaparecen las variables num1 y num2** correspondientes a los parámetros de la función add y ahora la variable res almacena el resultado de la función

0x0000000000	**num1 = 10**
0x0000000004	**num2 = 20**
0x000000000C	**res = 30**
0x000000000D	
0x000000000E	
0x000000000F	
0x0000000010	
0x0000000011	
0x0000000012	
0x0000000013	
0x0000000014	

Te diste cuenta de lo que comentamos en algún capitulo atrás “la memoria estática la administra el sistema operativo (la reserva y la libera automáticamente)”. ¿Cómo sabe el compilador que ya no va a usar las variables/atributos de la función **add** para liberarlas? Muy sencillo, gracias a los brackets/braces que delimitan la función 😊

Bueno, ahora que viste lo que sucede en memoria, viene la explicación teórica:

Las funciones se reservan en un espacio especial de la memoria “TEXT” segment, pero NO todo está guardado en éste segmento, sólo las instrucciones. Las variables y atributos que no son apuntadores se guardan en el stack y sí, cada que ejecutas una función que no tiene apuntadores se crean copias de los valores que le pasas a la función (los parámetros). Esto se llama **“Pasar parámetros por valor”.** Lo malo de pasar los parámetros así es que se copian, lo que significa que en memoria hay 2 veces el mimos valor repetido ☹ ¿esto no es ineficiente? Depende de lo que quiera el programador. Ej. Imagina que vas a a comprar un auto y te piden que dejes las escrituras de tu casa para garantizar el pago ¿les das tus escrituras? Noooo, claro que no, pq pueden hacer mal uso de tus escrituras, entonces lo que les das es UNA COPIA para que puedan hacer lo que tengan que hacer, pero sin darles tus valores originales. Así mismo en programación, tú decides si quieres que dar copia de tus datos/variables o les das los valores originales.

Hasta Aquí llegamos en la sesión 1 😊

Pasar parámetros por referencia.

La segunda forma de pasar parámetros a una función es por referencia ¿esto no te trae un recuerdo? Sí, hablamos de referencia cuando hablamos de direcciones de memoria y sí, cuando pasamos un valor por referencia, significa que en lugar de copiarlo pasamos la dirección implícita de la variable que pasamos. También es importante qué recuerdes ¿qué operador usábamos para obtener la dirección de una variable? Sí, usábamos el ampersand **&** . Veamos un ejemplo de cómo usar el ampersand para pasar parámetros por referencia

```
int add(int &num1, int &num2)
{
    return num1+num2;
}

int multiply(int &num1, int &num2)
{
    return num1*num2;
}

int main2()
{
    int num1 = 10;
    int num2 = 20;

    int res = add(num1, num2);
}
```

Pasar por referencia es tan simple como agregarle el **&** en la lista de parámetros de la función. Todo lo demás es idéntico al código que tenemos en la sección anterior. Y claro, YA NO HAY COPIAS INECESARIAS EN MEMORIA.

¿cómo sé que el & es para pasar por referencia y no para obtener la dirección de una variable?

Fácil. Si el ampersand está en la firma de una función, entonces es paso por referencia, si no, entonces es para obtener la dirección de una variable

Te acuerdas ¿qué hacías cuando necesitabas que una variable la pudieran ver diferentes funciones?

Sí, te ponías a declarar VARIABLES GLOBALES. Tache, tache, tache. Es una muy mala práctica utilizar variables globales. Ahora que ya conoces cómo pasar valores por referencia entonces no necesitas usar variables globales 😊

A continuación un ejemplo en el que se muestra qué pasa si cambias el valor de un parámetro que has pasado por referencia y otro que ha pasado por valor/copia

```
void cambiaValores(int& num1, int num2)
{
    //  ahora cambiemos los valores de ambos parámetros
    //  y luego ve que sucede en la función main
    num1 = 111111;
    num2 = 222222;
}
int main()
{
    int num1 = 1;
    int num2 = 2;

    cout<<"num1 antes de llamar la función cambiaValores vale " << num1 << endl;
    cout<<"num2 antes de llamar la función cambiaValores vale " << num2 << endl;

    cambiaValores(num1, num2);

    cout<<"num1 antes de llamar la función cambiaValores vale " << num1 << endl;
    cout<<"num2 antes de llamar la función cambiaValores vale " << num2 << endl;
}
```

num1 pasa por referencia

Y **num2** pasa por valor

Pasar parámetros por dirección

La última forma de pasar parámetros es por dirección ¿otra vez direcciones? Sí, ajajajaj, sufran!!!!!!!!!

Bueno, cuando hablamos de direcciones, hablamos de **apuntadores** y sí, pasar parámetros por dirección significa pasar parámetros como apuntadores 😊. Veamos un ejemplo y vayamos explicando paso a paso cada instrucción y viendo qué sucede en memoria.

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

void modify(int* value)
{
    *value = 1000;
}
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo en la función se modifica la variable value: " << value;
}
```

Veamos qué pasa en memoria cuando ejecutamos el programa anterior

```
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo
}
```

0x0000000000	value = 8
0x0000000004	
0x0000000008	
0x000000000C	
0x0000000010	
0x0000000014	
0x0000000018	
0x000000001C	
0x0000000020	
0x0000000022	
0x0000000024	

```
void modify(int* value)
{
    *value = 1000;
}
```

0x0000000000	value = 8
0x0000000004	value = 0x00000000
0x0000000008	
0x000000000C	
0x0000000010	
0x0000000014	
0x0000000018	
0x000000001C	
0x0000000020	
0x0000000022	
0x0000000024	

```
void modify(int* value)
{
    *value = 1000;
}
```

0x0000000000	value = 1000
0x0000000004	value = 0x00000000
0x0000000008	
0x000000000C	
0x0000000010	
0x0000000014	
0x0000000018	
0x000000001C	
0x0000000020	
0x0000000022	
0x0000000024	

Sí, así como se ve en el diagrama, cuando pasamos parámetros por dirección es posible modificar los valores como y **se modifican los valores originales** (esto mismo pasa cuando pasas por referencia).

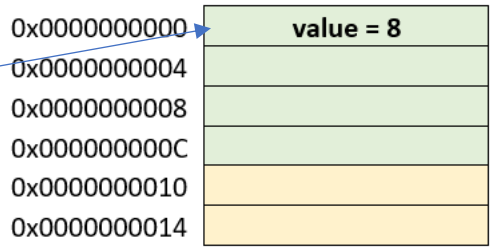
Pero ¿qué pasaría si modificaras el apuntador dentro de la función? No, no me refiero a cambiar el valor al que apunta el pointer (eso lo hicimos en el ejercicio pasado, me refiero a cambiar la dirección a la que apunta?

Veamos un ejemplo y sigamos paso a paso lo que sucede en memoria.

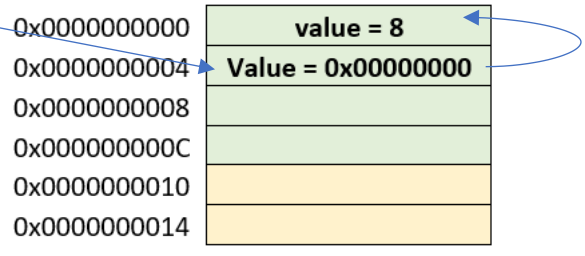
```
void modify(int* value)
{
    // ahora cambiamos la dirección a la que apunta el pointer
    value = new int;
    *value = 1000;
}
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo en la función se modifica la variable value: " << value;
}
```

Ahora veamos qué pasa en memoria paso a paso

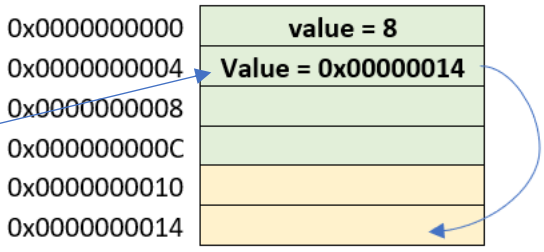
```
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo
}
```

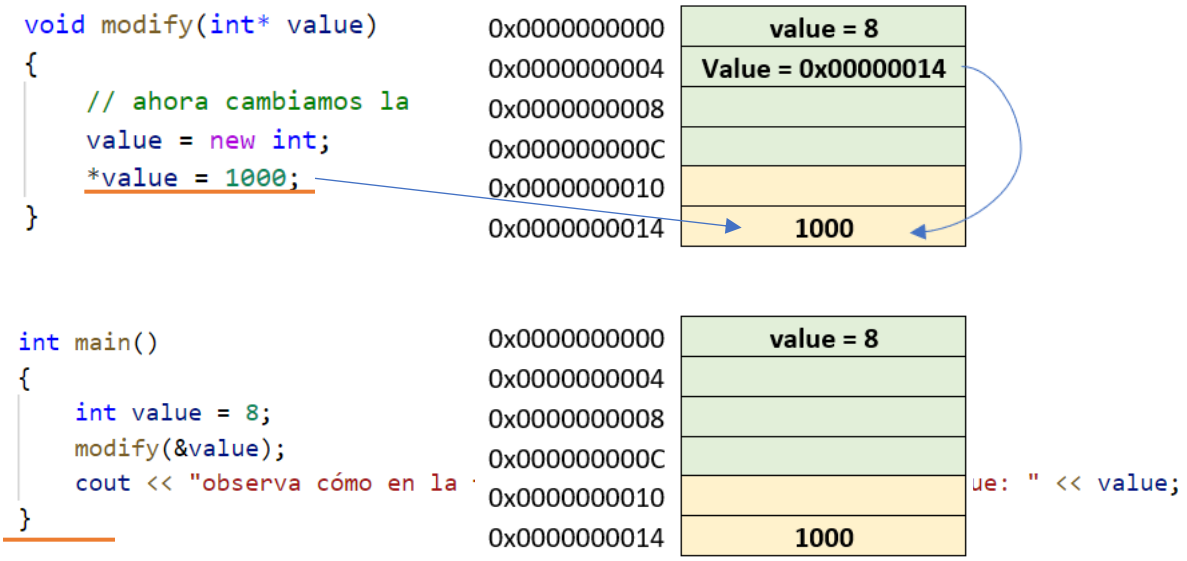



```
void modify(int* value)
{
    // ahora cambiamos la
    value = new int;
    *value = 1000;
}
```




```
void modify(int* value)
{
    // ahora cambiamos la
    value = new int;
    *value = 1000;
}
```





¿Ves claramente cuál es el problema de modificar el apuntador? (**no es lo mismo modificar el apuntador que modificar el valor al que apunta**)

Bueno, en el ejemplo anterior se observa claramente que cuando pasas por dirección puedes modificar los valores como si fueran los originales, pero sí modificas la dirección del apuntador, entonces resulta en un error llamado "**memory leak**". Ahora sí, pregunta de los 100 millones ¿cómo harías para poder modificar la dirección del apuntador sin que se generara un memory leak? Podrías obtener hasta 5 puntos de la calificación final si lo resuelves SIN AYUDA (J, tnp).

Programación Orientada a Objetos

Los temas que vamos a revisar son los siguientes

- Introducción. Definiciones básicas
-

Introducción

Existen diferentes paradigmas de programación.

- Estructurada. Como la que aprendiste en los módulos anteriores de este manual o lo que aprendiste el semestre pasado en Python
- Orientada a Objetos.
- Funcional. Orientada originalmente a cómputo científico (ahora ya es de propósito general)
- Scripting

Nos enfocaremos en el paradigma orientado a objetos ya que, además de ser muy popular, es muy fácil modelar problemas porque se asocian a estructuras familiares para el ser humano (Objetos).

La POO se creo para facilitar el modelado de la realidad en la que vivimos. Para lograrlo es necesario ir de lo general a lo particular, es decir, se necesita una forma de categorizar todo lo que existe en el mundo real (o cualquier otra cosa). Por ejemplo:

1. Podemos hablar de manera general de COMPUTADORAS. Es una generalización porque existen diferentes tipos de computadoras (con más capacidad de MEMORIA, PROCESADOR, TAMAÑO, COLOR, etc.)
2. Hablar de PERSONAS es también una generalización porque existen diversas características que hacen diferentes a las personas (genero, raza, color de piel, etc.)
3. Podemos pensar en los CLIENTES de las empresas como una generalización de aquellas PERSONAS / EMPRESAS que compran un PRODUCTO o SERVICIO

Observa en los ejemplos anteriores que todo lo que está en mayúsculas se refiere a una generalización de algo. Esa generalización se llama CLASES. Pero ¿por qué son importantes las generalizaciones? Porque es más fácil referirnos a algo general que la mayoría comprende a referirnos a los detalles particulares de cada cosa. Piensa en lo siguiente: Qué me dirías si te pregunto ¿de qué trata el manual que les compartí? ¿entrarías en detalles o me dirías lo más general? Si tú me lo preguntaras a mí te diría simplemente C++. Las generalizaciones permites hablar de las cosas sin conocer los detalles particulares de las mismas.

Por supuesto no vivimos en un mundo donde sólo existan generalidades por lo que tenemos que ir a los detalles. Pensemos en los ejemplos de clase que dimos arriba. ¿Qué particularidades me puedes decir de cada una de ellas?

1. Si fuera mi Computadora te diría que tiene 16gb de RAM, un disco de estado sólido de 256gb, con un procesador Core i5 de 8va generación y una tarjeta gráfica Intel.
2. Si hablara de la Persona que es el presidente de la república te diría que es una persona de edad avanzada, de estatura promedio, que habla lentamente, etc.

3. Si me preguntas de mi cliente “X” te diría que compra regularmente productos de tipo “A”, que sus compras son a finales de mes y que siempre pide un descuento

¿Te diste cuenta que pasamos de lo general a lo particular? Cuando hablamos de las cosas de manera particular nos referimos a un OBJETO. Los objetos son aquellos que tienen valores que, en su conjunto, los hacen únicos.

Algunos ejemplos de atributos que hacen únicos a los objetos pueden ser: edad, peso, género, nombre, número de hermanos, salario mensual, curp, matrícula, nómina, folio. Todas estas características hacen únicos a los objetos. Estas características en POO se llaman ATRIBUTOS.

Ahora que podemos ir de lo general (CLASE) a lo particular (OBJETO) a través de (ATRIBUTOS) es necesario agregar algo que le de vida a las cosas. El ejemplo más básico es la clase PERSONA. Qué hace que la persona tenga vida? Sus acciones. Las acciones de una clase se llaman MÉTODOS. Dichas acciones son las que permiten la interacción entre objetos.

Algunos ejemplos de métodos son: caminar, correr, cobrar, pagar, hablar, cantar, calcular, sumar, discutir, **reprobar,** quejarse, etc. Todas estas son acciones que pueden realizar algunos de los objetos de los que hablamos en los párrafos anteriores.

Finalmente es que podemos determinar los elementos fundamentales de la programación Orientada a Objetos

- Clases
- Objetos / Instancias. Otro nombre por el que se conocen los objetos es INSTANCIA.
- Atributos
- Métodos

Listo, fin del curso. No, lo importante de estos elementos es ¿cómo logro que se interrelacionen entre ellos?

Bueno, todo con calma. Vayamos al código y veamos cómo se define en programación los elementos básicos de la POO.

Las clases se definen con la palabra reservada class. El scope de las clases, así como de las funciones se especifica con los brackets/braces. Ej.

```
class Persona
{

}karla;
```

Nota que al finalizar el scope de la clase Persona coloqué el nombre “karla” que significa que estoy creando una clase (Persona) y un objeto (karla) a la vez. Sin embargo, como dijimos más arriba del texto, los objetos son entidades específicas y son específica gracias a los valores de sus atributos, pero en ésta clase no le he colocado ningún atributo, por lo que es preferible NO crear el objeto. Así, la creación de mi clase quedaría como sigue:

```
class Persona
{

};
```

Ahora vamos a agregarle atributos a nuestra clase. Los atributos son características a los que le puedo asignar valores (nombre, dirección, teléfono) y que me permiten identificar a una persona. Dado que son valores, entonces siguen las reglas de las variables en C++ (tipos de datos ya sean nativos o que sean definidos por el usuario)

```
class Persona
{
    string nombre;
    string direccion;
    string telefono;
};
```

Observa que los atributos son VARIABLES a las que les puedo asignar valores. Si son variables, entonces pueden ser APUNTADORES

Hasta aquí todo parece que va bien, sin embargo piensa en una persona que tiene nombre, dirección y teléfono, dicha persona puede cambiar el valor de los atributos, pero alguien externo podría? No, no debería. Para evitar que alguien ajeno a la persona cambie el valor de sus atributos C++ provee de lo llamado MODIFICADORES DE ACCESO. Hay tres tipos de modificadores de acceso:

- public. Cualquier externo a MÍ puede cambiar el valor de mis atributos públicos
- private. Sólo yo puedo cambiar el valor de mis atributos (no quiere que venga cualquiera y me cambie de nombre sin mi permiso)
- protected (lo aprenderemos la siguiente sesión)

Veamos cómo se ven los modificadores de acceso en el código

```
class Persona
{
private:
    string nombre;
    string direccion;
    string telefono;
public:

protected:

};
```

Ya que tenemos al clase y sus atributos, pero sin valores ¿cómo programamos un objeto/instancia? Muy simple, pero antes piensa: C++ no tiene algún tipo de variable/tipo de datos que me permita guardar a una persona ☹ pero, podría usar una clase como si fuera una variable/tipo de datos definido por el usuario? Correcto, así se crean las instancias.

```
int main()
{
    Persona pedro;
    Persona pablo;
    Persona paco;
}
```

Observa que las instancias / objetos se definen igual que se definen las variables

TipoDeDatos nombreVariable; pero ahora

Clase nombreObjeto;

Sí, observa el código anterior y verás que las Clases son "user defined data types". Dado que las clases son como las variables, entonces siguen exactamente las mismas reglas para su creación (incluyendo las de APUNTADORES)

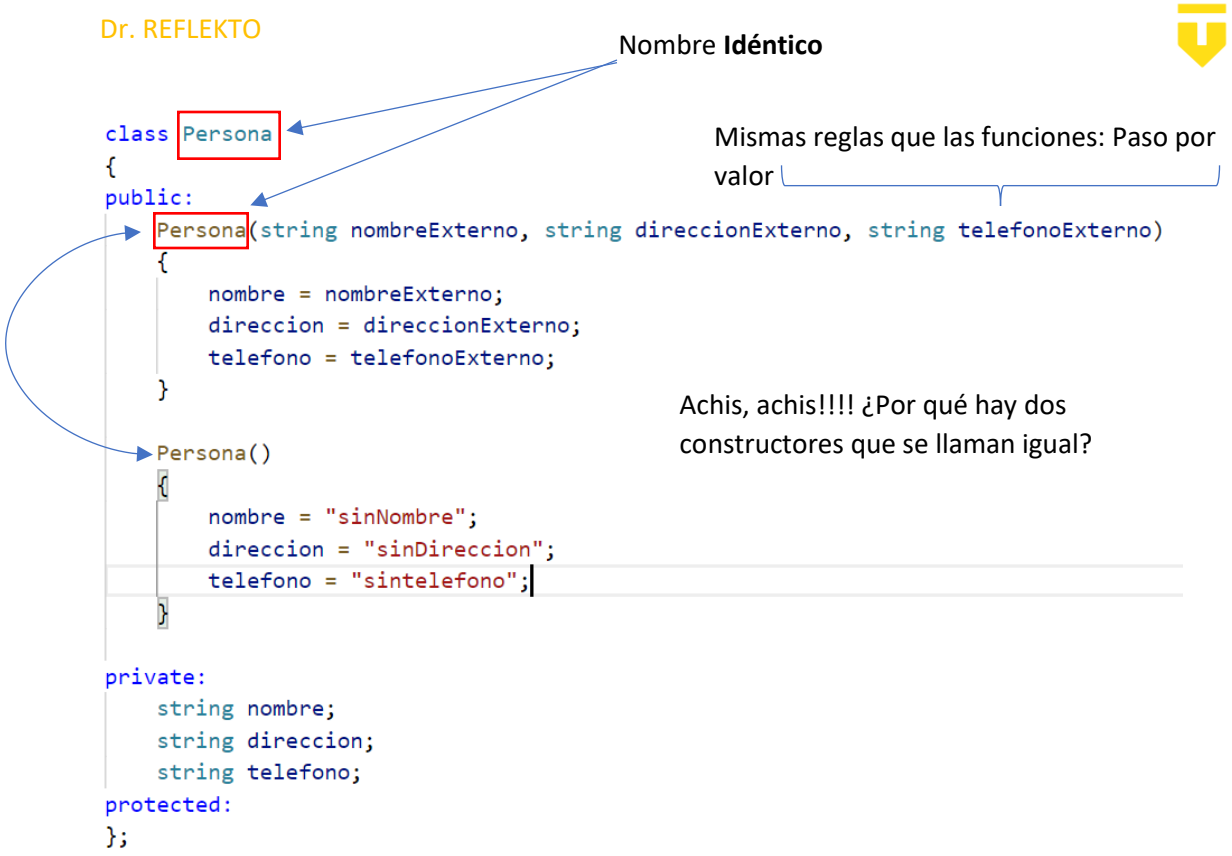
Lo único que nos falta es que nuestra instancia tenga valores (para que sean únicos -así como está mi programa no puedo identificar quien es pedro, pablo o paco). Para asignar valores a los atributos nos basaremos en la realidad: poco después de cuando naciste te pusieron tu nombre, apellidos, género, etc. Haremos lo mismo en programación y para eso agregaremos un elemento nuevo: CONSTRUCTORES.

CONSTRUCTORES

Un constructor es un método que permite construir un objeto (la construcción incluye la **asignación de valores** a los atributos, dicha asignación también es llamada inicialización). Sí, hablamos arriba que los métodos son las acciones que dan vida a los objetos, pero no hemos dicho cómo se crean los métodos. Es muy sencillo y ya sabes cómo hacerlo: LOS MÉTODOS SON FUNCIONES DE UNA CLASE. Sí, las funciones que ya sabes usar pueden ser métodos de clase (y en consecuencia siguen las mismas reglas que las funciones tradicionales: paso por valor, por referencia y por dirección). Los constructores son funciones/métodos que tienen sus reglas únicas de declaración/definición:

- Se debe llamar (su nombre) EXACTAMENTE IGUAL QUE LA CLASE
- NO TIENE VALOR DE RETORNO. Esto es una gran diferencia con respecto a las funciones que siempre tienen valor de retorno (void es un valor de retorno, de hecho es un tipo de datos 😊)
- El constructor se ejecuta automáticamente cuando creamos una instancia de clase (un objeto -ve el código anterior-)
- Si no definimos / declaramos un constructor, entonces C++ crea uno por default

Veamos en código un constructor



Observa en el código anterior que HAY DOS CONSTRUCTORES. ¿Cómo sé que son dos constructores? Porque siguen las reglas básicas de un constructor (mismo nombre de la clase y sin valor de retorno). El primer constructor es un método que tiene argumentos a partir de los cuales se guardan los valores de los atributos de la clase, mientras que el segundo es para el caso en que al programador se le olvide que tiene que dar valores a los atributos. El segundo se llama **constructor default** y sirve para asignar valores default a los atributos de un objeto. El primero se llama **constructor con parámetros**

Tipos de costructores (los revisaremo más adelante):

- Default Constructor
- Parameterized constructor
- Copy constructor

ACCESO A LOS COMPONENTES (atributos y métodos) DE UNA CLASE

Para acceder a los métodos o atributos públicos de una clase se utiliza el operador “punto”. Veamos un ejemplo:

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

class User
{
private:
    string userName;
    string password;
public:
    User(string theUserName, string thePassword)
    {
        userName = theUserName;
        password = thePassword;
    }
    string getUserName()
    {
        return userName;
    }
};

int main()
{
    User selena("selena", "12345");
    selena.getUserName();
}
```

Utilizamos el punto para acceder a los componentes públicos de la clase

APUNTADORES A CLASES e INSTANCIAS (de-referenciar)

Aquí vamos a explicar cómo crear apuntadores a instancias de clases y cómo puedes de-referenciar clases que son apuntadores. Hasta ahora sólo has aprendido una forma de de-referenciar apuntadores usando el operador *, pero no es la única forma de poder de-referenciar un apuntador. Hay 3 formas diferentes:

1. Utilizando el operador *****
2. Utilizando el operador **[]**
3. Uitilizando el operador **->**

Las primeras dos formas de de-referenciar son equivalentes y se pueden intercambiar dependiendo de la preferencia del usuario. La tercer forma de de-referenciar es exclusiva para objetos (permite acceder a los atributos o métodos públicos de una clase). En el siguiente ejemplo se muestra cómo se usan los tres operadores de de-referenciación.

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

class Invoice
{
    double amount;
    string clientName;
public:
    void setAmount(double theAmount)
    {
        amount = theAmount;
    }
    double getAmount()
    {
        return amount;
    }
};

int main()
{
    Invoice* marchInvoice = new Invoice;
    marchInvoice->setAmount(10000);

    cout << (*marchInvoice).getAmount() << endl;
    cout << marchInvoice->getAmount() << endl;
    cout << marchInvoice[0].getAmount() << endl;
}
```

Apuntador a una clase y reserva de memoria dinámica con new

1. Operador *
2. Operador ->
3. Operador []

Relaciones entre Clases

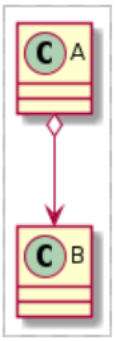
Ahora que ya sabemos cómo crear, instanciar y acceder a un objeto es necesario que sepamos cómo podemos interactuar con otros objetos. Esta forma de interacción entre clases y objetos se llama “relaciones entre clases”.

- Agregación
- Asociación
- Composición
- Herencia

Agregación

La clase A forma parte de la clase B. Lo más importante de esta relación es que la creación y destrucción de la clase B es independiente de A (A no tiene ninguna responsabilidad de crear o

destruir a B. Se representa con un “diamante” sin relleno el diamante sin relleno va del lado del todo (La clase A se considera “el todo” o el objeto principal

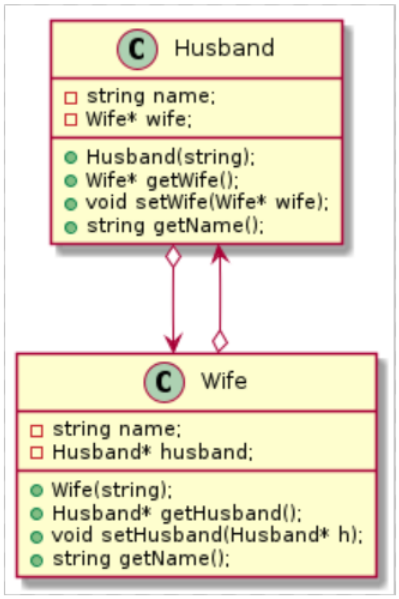


Ejemplo: La clase Husband tiene como atributo a la clase Wife, es decir, está compuesta por una Wife, pero no tiene la responsabilidad de crearla ni destruirla. A su vez la clase Wife está compuesta por un Husband, pero no tiene la responsabilidad ni de crearla ni de destruirla.

```
class Husband
{
    string name;
    Wife* wife;
public:
    Husband(string);
    Wife* getWife();
    void setWife(Wife* wife);
    string getName();
};
```

```
class Wife
{
    string name;
    Husband* husband;
public:
    Wife(string);
    Husband* getHusband();
    void setHusband(Husband* h);
    string getName();
};
```

El **DIAGRAMA DE CLASES** de la relación entre Wife y Husband es el siguiente:



Asociación

La clase A usa un miembro de la clase B. En esta relación la clase A no está compuesta por la clase B, sólo utiliza un elemento de B. Se representa por una línea con flecha (algunos autores sólo colocan la línea sin flechas)

Ejemplo:

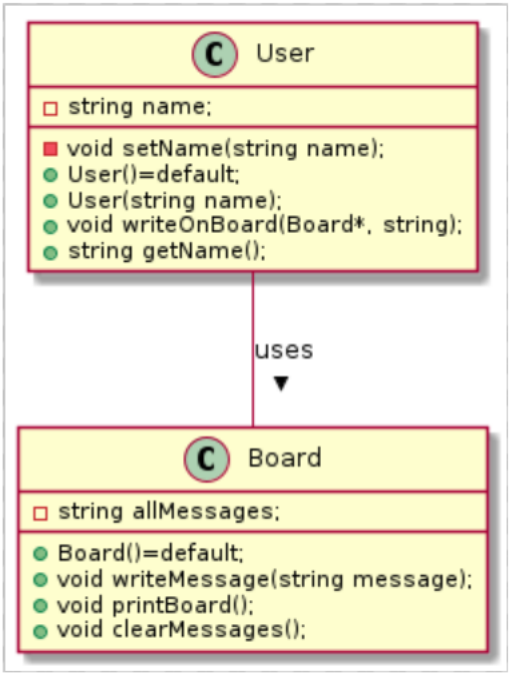
Te piden que como programador desarrolles un programa que permita escribir mensajes en un pizarrón. Para ello creas 2 clases: User y Board. Dado que el user quiere escribir en el Board, entonces se dice que el User **usa** a la clase Board. Sólo la usa porque en su lista de atributos de user NO HAY UN BOARD. Board únicamente está como parámetro en el método writeOnBoard(Board*,string). A continuación los encabezados de ambas clases

```
class User
{
    void setName(string name);
    string name;
public:
    User()=default;
    User(string name);
    void writeOnBoard(Board*, string);
    string getName();

    ~User();
};
```

Aquí se usa a la clase Board y NO HAY UN BOARD en la lista de atributos de la clase

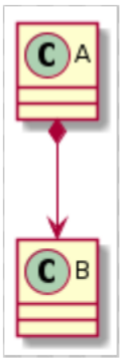
```
class Board
{
    string allMessages;
public:
    Board()=default;
    void writeMessage(string message);
    void printBoard();
    void clearMessages();
};
```

A Continuación el **diagrama de clases** de Board y User



Composición

La clase A está compuesta por la clase B. Lo importante de "estar compuesto" implica que la responsabilidad de creación y destrucción de B recae por completo en A. La clase A es responsable de B y la clase B **no puede existir** fuera de A. La composición se representa por un rombo con relleno.



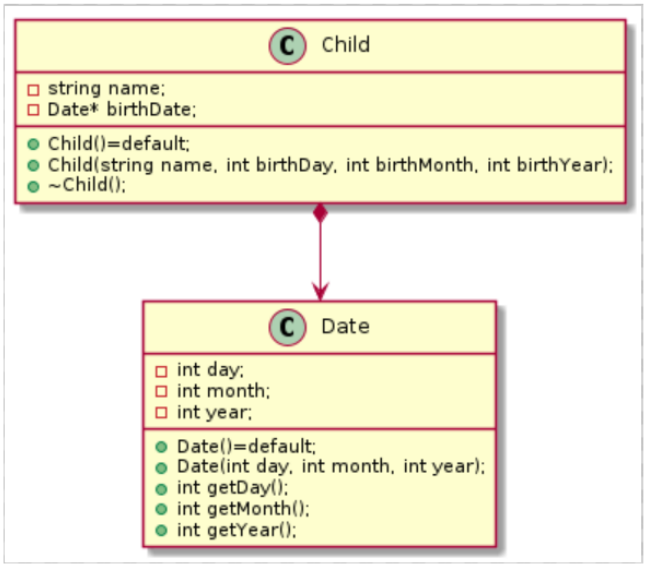
Ejemplo:

Suponga que le piden modelar una clase Child. Dicha clase tiene dentro de su lista de atributos la clase Date que permite especificar la fecha de nacimiento del mismo. Cuando se crea al niño se tiene la responsabilidad de crear la fecha de nacimiento y guardarla. Cuando la clase Child deje de existir, entonces también se deberá destruir la instancia de su fecha de cumpleaños. Como puedes observar, a diferencia de la agregación, aquí se deben construir ambos objetos al mismo tiempo y "la vida útil" del sub-objeto (Date) dependerá de la vida de la clase principal. A continuación los encabezados de ambas clases:

```
class Date;
class Child
{
    string name;
    Date* birthDate;
public:
    Child()=default;
    Child(string name, int birthDay, int birthMonth, int birthYear);
    ~Child();
};
class Date
{
    int day, month, year;
public:
    Date()=default;
    Date(int day, int month, int year);
    int getDay();
    int getMonth();
    int getYear();
};
Child::Child(string name, int diaNacimiento, int mesNacimiento, int anioNacimiento)
{
    this->name = name;
    birthDate = new Date(diaNacimiento, mesNacimiento, anioNacimiento);
}
Child::~Child()
{
    delete birthDate;
}
```

En la relación de composición la clase principal tiene la responsabilidad de **crear** y **destruir** a la subclase que la compone

El **diagrama de clases** para Las clases anteriores es el siguiente



Herencia

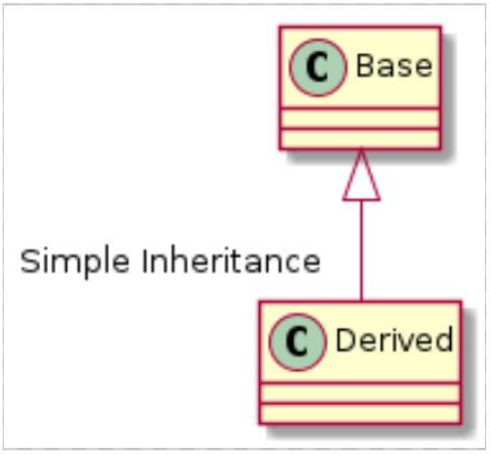
La relación de herencia es muy semejante a la herencia entre padres e hijos: los padres “te transmiten” características de ellos “la voz se la heredó a su papá”. En programación la herencia tiene muchas ventajas, pero las más relevantes son:

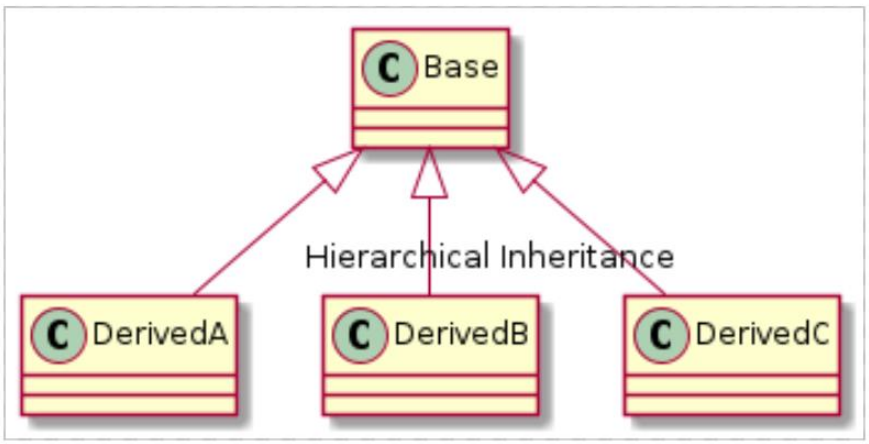
- Rehúso de código.
- Agregar nuevas características a una clase. Una clase es “cerrada” porque no permitimos que nadie la modifique (encapsulamiento), pero ¿cómo hacemos para agregar características sin modificar las clases que ya hemos hecho? Con herencia

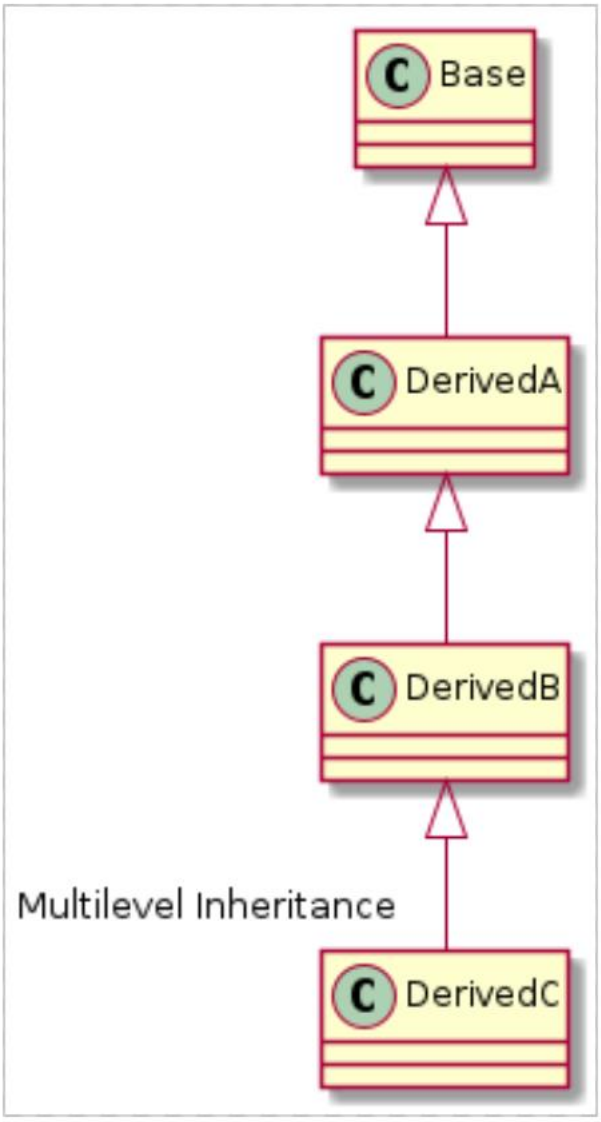
La herencia es una relación del tipo “es un”, esto significa que si la clase B hereda de la clase A, entonces la clase B es un subconjunto de la clase A. Se dice que la clase B también es la clase A (algo así como doble identidad). La herencia está representada por una flecha donde la terminación es abierta.

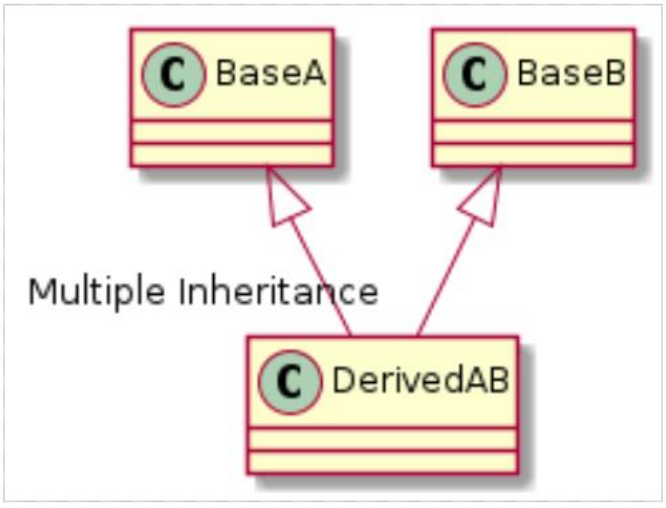
Existen diferentes tipos de herencia.

- Simple
- Multinivel
- Jerárquica
- Múltiple






Base
DerivedA
DerivedB
Multilevel Inheritance
DerivedC


BaseA
BaseB
Multiple Inheritance
DerivedAB

Herencia y Modificadores de acceso

Los modificadores de acceso no sólo se utilizan para atributos y métodos, sino también para la relación de herencia. A continuación se coloca una tabla de qué pasa con los atributos de las clases derivadas cuando se utilizan los modificadores de acceso (public, private, protected).

ACCESSIBILITY IN PUBLIC INHERITANCE

Accessibility	Private Members	Protected Variables	Public Variables
Derived class	Not accessible	Protected	Public
2nd Derived class	Not accessible	Protected	Public

ACCESSIBILITY IN PROTECTED INHERITANCE

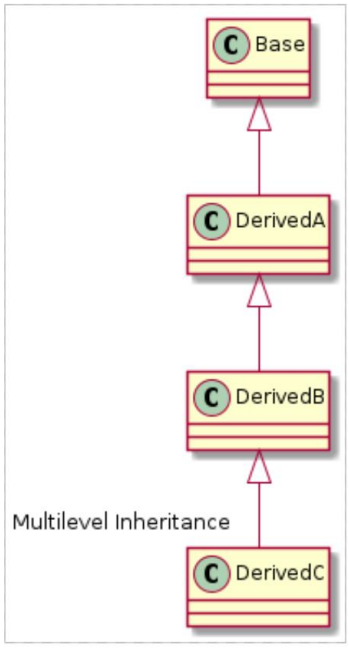
Accessibility	Private Members	Protected Variables	Public Variables
Derived class	Not accessible	Protected	Protected
2nd Derived class	Not accessible	Protected	Protected

ACCESSIBILITY IN PRIVATE INHERITANCE

Accessibility	Private Members	Protected Variables	Public Variables
Derived class	Not accessible	Private	Private
2nd Derived class	Not accessible	Not accessible	Not accessible

¿Cómo funciona la herencia de manera interna? Construcción de clases Derivadas (hijas)

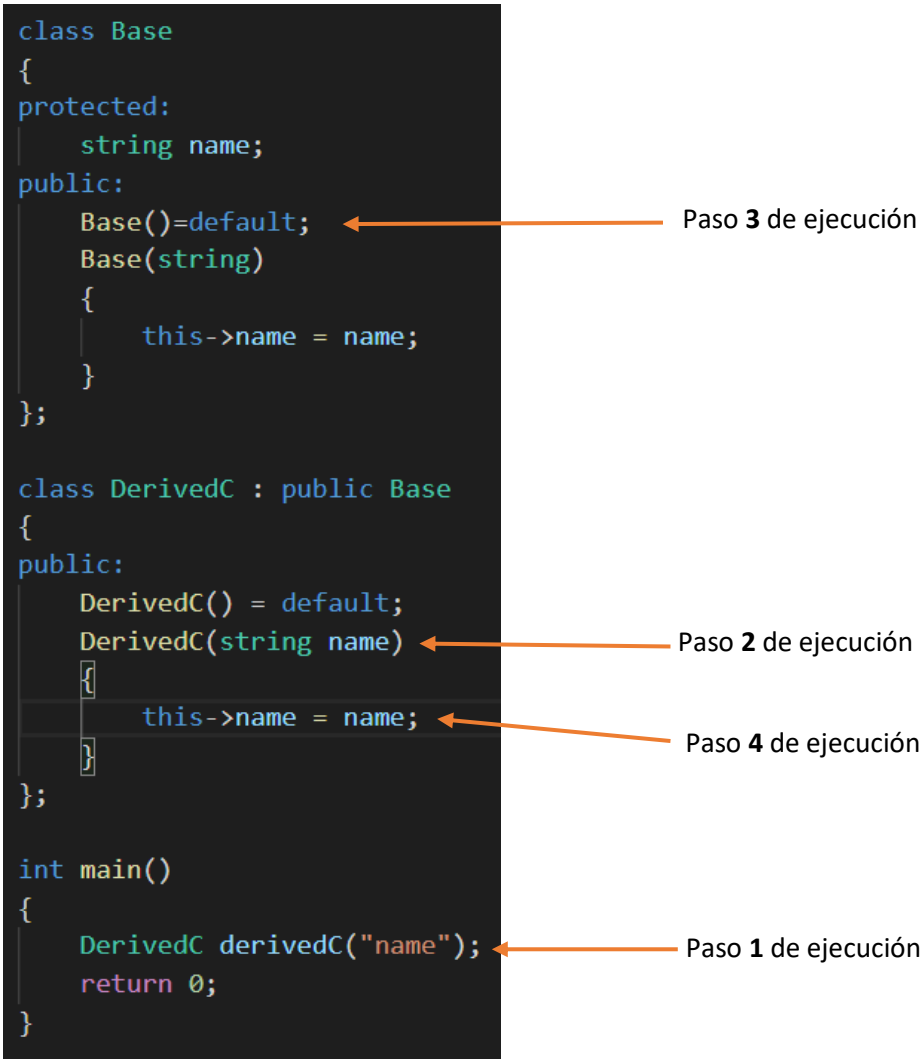
Ahora ya sabes que cuando una clase hereda de otra, entonces **comparte** todos sus atributos, públicos y protegidos (revisas modificadores de acceso de herencia). Para que esto suceda es importante que sepas que cuando creas una instancia de la clase Derivada, de manera interna se crean 2 instancias en memoria: la instancia de la clase derivada y la instancia de la clase base. Dada la afirmación anterior: ¿cuántas y cuáles instancias hay en memoria cuando se crea un instancia de la clase “DerivedC”?




```
int main()
{
    DerivedC derivedC;
    return 0;
}
```

Correcto: las instancias en memoria son 4. (la instancia de DerivedC, la de DerivedB, la de DerivedA y la instancia de Base. Pero ¿cuál es el orden interno de construcción de las clases?

En una jerarquía de clases heredadas la primer clase que se crea es la clase Base y a partir de ella se van creando cada una de las hijas derivadas (tiene sentido porque cómo podríamos compartir un atributo de la clase Base si la clase Base no existe). Esta afirmación nos hace reflexionar ¿cómo se mandan llamar los constructores? ¿si se crea primero la clase Base, entonces el siguiente código no es correcto? Sí es válido el código, pero observa el orden de ejecución de las instrucciones



Como puedes ver la instrucción `this->name = name;` es la última en ejecutarse. Esto es válido, pero NO CORRECTO ¿te imaginas por qué no es correcto?

Es incorrecto porque cuando creamos la instancia de derived enviamos un parámetro (name) esperando que se usara el constructor que recibe un parámetro y se asignara el valor de name, pero no sucede así ya que en el paso **3** se manda llamar el constructor por default que crea una instancia vacía de name y luego, hasta el paso **4** esa instancia vacía es sobre escrita usando la instancia que

mando el usuario. El problema es que hay una instancia adicional del atributo name (sí, la instancia default). Esto podría no parecer un problema, pero imagina que fuera un objeto que tiene muchos atributos (objeto complejo), entonces, en lugar de directamente copiar el valor que nos da el usuario, tendríamos una copia "default" de dicho objeto y luego ya la sobre-escribiríamos.

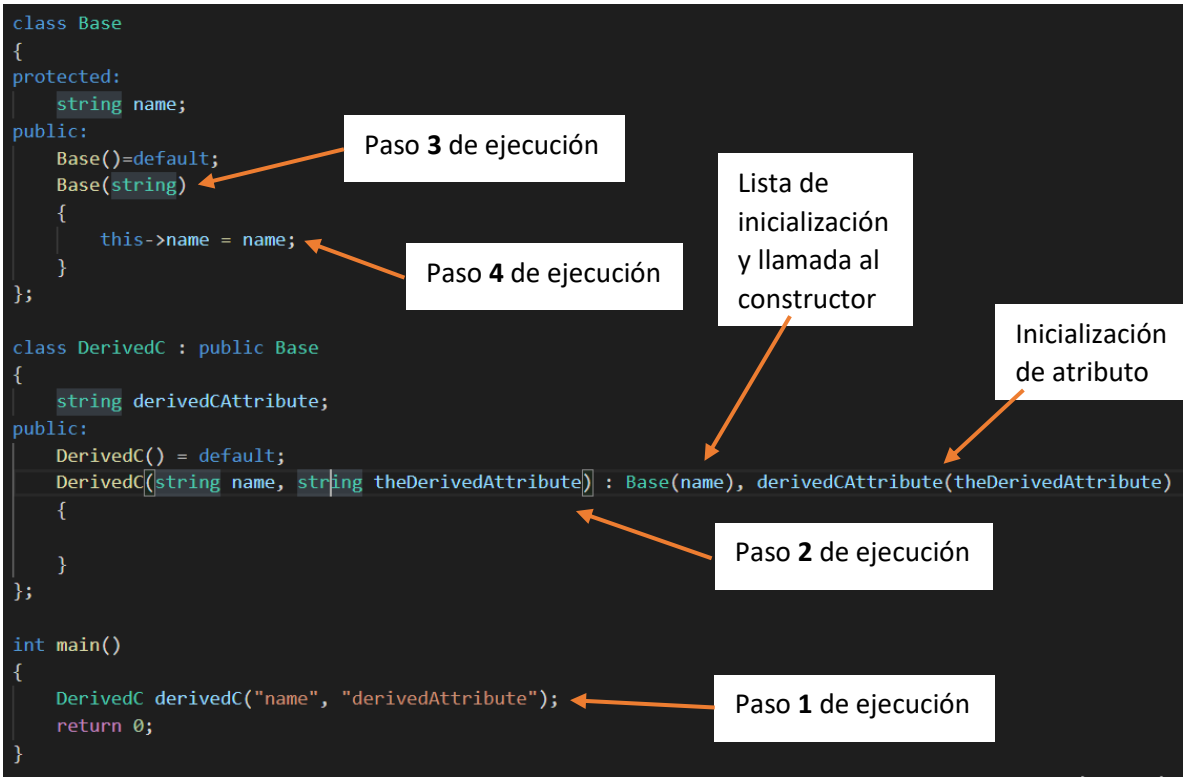
Para corregir el error es necesario utilizar el constructor del padre para hacer la asignación directamente. Aquí es importante notar lo siguiente:

- Los **constructores** y **destructores** NO SE HEREDAN

Bueno, una vez puntualizado lo anterior es necesario encontrar un punto entre el paso 2 y 3 en el que se permita utilizar el constructor de la clase base y evitar que haya copias "default" que no son necesarias y que consumen recursos. Ese punto está justo en el paso 2 con la llamada "**Lista de inicialización**". La lista de inicialización permite inicializar valores antes de la ejecución del cuerpo del constructor (que es justo lo que queremos, poder llamar al constructor de la clase padre antes de la ejecución del cuerpo del constructor. Usar las listas de inicialización es muy simple:

- Colocar **:** después de los parámetros del constructor
- Asignar los valores

A continuación se corrige el código anterior y se muestra el uso de listas de inicialización (se agregó un atributo para la clase derivada de manera que puedas ver que también se pueden inicializar atributos de la misma clase)



Herencia y Métodos Heredados

Los métodos de las clases tienen un comportamiento particular. Veamos un ejemplo: