



C++ SEMANA 2

# Smart pointers (punteros inteligentes)

Como hemos visto, la asignación dinámica de memoria conlleva varios riesgos: punteros colgantes, fugas de memoria, etc. Es un tema que debemos explorar obligatoriamente debido a que puede encontrar este tipo de asignación en código existente y deberá saber como lidiar con el.

En C++ moderno, existen otros mecanismos que nos aseguran la correcta liberación de memoria reservada dinámicamente. Esto nos lleva a los Smart pointers. Estos punteros (*Smart*) *imitan el comportamiento de los punteros crudos (raw pointers), pero proporcionando algoritmos de administración de memoria*. En palabras simples automatizan la liberación de memoria.

Un puntero inteligente contiene un puntero integrado, definido como un template class cuyo parámetro de tipo es el tipo del objeto apuntado, por lo que puede declarar punteros inteligentes que apunten a un objeto de cualquier tipo (primitivos, objetos...).

# Smart pointers (punteros inteligentes)

Un objeto `unique_ptr` envuelve un puntero crudo (raw pointer) y es responsable de su tiempo de vida. Cuando este objeto se destruye, en su destructor elimina el puntero crudo asociado liberando la memoria reservada.

Los tipos de punteros inteligentes están definidos por plantillas dentro del encabezado `memory` de la Biblioteca estándar, por lo que debe incluir este header en su archivo fuente para usarlos. Hay tres tipos de punteros inteligentes, todos definidos en el espacio de nombres estándar:

# Smart pointers (punteros inteligentes)

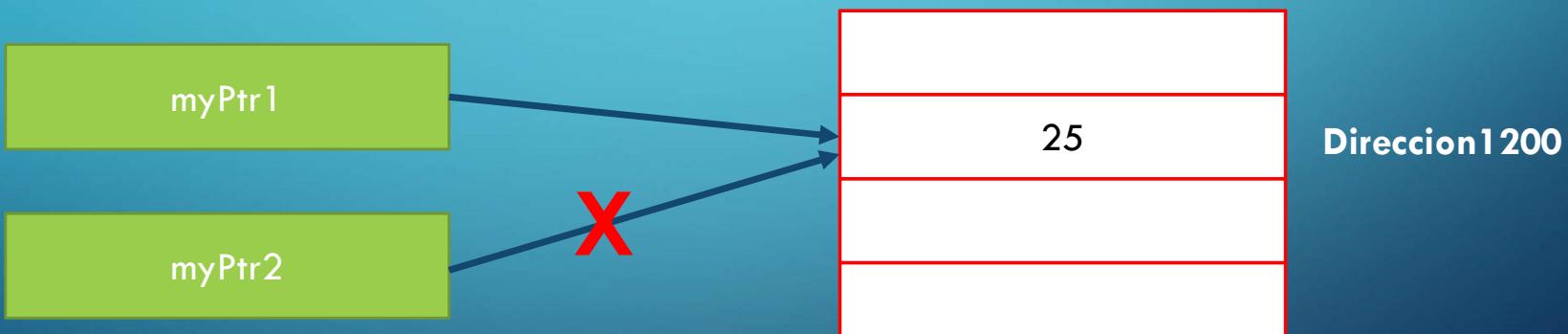
- Un objeto `unique_ptr<T>` se comporta como un puntero al tipo `T` y es "único" en el sentido de que solo puede haber un único objeto `unique_ptr<>` que contenga la misma dirección. En otras palabras, nunca puede haber dos o más objetos `unique_ptr<T>` apuntando a la misma dirección de memoria al mismo tiempo. Se dice que un objeto `unique_ptr<>` posee aquello a lo que apunta exclusivamente. Esta singularidad se ve reforzada por el hecho de que el compilador nunca le permitirá copiar un `unique_ptr<>`.

# Smart pointers (punteros inteligentes)

Punteros `unique_ptr<T>`

```
std::unique_ptr<int>myPtr1 = std::make_unique<int>(25);
```

```
std::unique_ptr<int>myPtr2 = myPtr1;// Error!
```



# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

El objeto `unique_ptr<T>` almacena una dirección de forma única, por lo que el valor al que apunta pertenece exclusivamente al puntero inteligente `unique_ptr<T>`. Cuando se destruye el `unique_ptr<T>`, también se destruye el valor al que apunta. Como todos los punteros inteligentes, un `unique_ptr<T>` es más útil cuando se trabaja con objetos asignados dinámicamente. Entonces, los objetos no deben ser compartidos por múltiples partes del programa o donde el tiempo de vida del objeto dinámico está naturalmente ligada a la de un solo otro objeto en su programa. Un uso común para un `unique_ptr<T>` es contener algo llamado *puntero polimórfico*, que en esencia es un puntero a un objeto asignado dinámicamente que puede ser de cualquier tipo de clase relacionada.

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

En esta etapa utilizaremos Smart\_pointers con tipos fundamentales. La verdadera potencia de estos punteros la veremos cuando veamos objetos. Veamos como se define un `unique_ptr<T>`.

Antes de C++14:

```
std::unique_ptr<double> pdata {new double{999.0}};
```

C++14 introdujo la plantilla de función `std::make_unique<>()`, esta es la forma recomendada y la que debería utilizar.

```
std::unique_ptr<double> pdata { std::make_unique<double>(999.0) };
```

# Smart pointers (punteros inteligentes)

Usando punteros `unique_ptr<T>`

**RECOMENDACIÓN:** Para crear un objeto `std::unique_ptr<T>` que apunte a un valor T recién asignado, utilice siempre la función `std::make_unique<T>()`. no solo es más corto (siempre que use la palabra clave `auto` en la definición de variable), esta función también es más segura contra algunas fugas de memoria más sutiles

# Smart pointers (punteros inteligentes)

Solo un puntero `unique_ptr` posee un objeto a la vez. Es decir solo puede existir un `unique_ptr` que apunta a una dirección determinada. Es decir, lo siguiente no está permitido:

```
std::unique_ptr<int>myPtr1 = std::make_unique<int>(25);  
std::unique_ptr<int>myPtr2 = myPtr1;// Error!
```

Podemos utilizar la semántica de movimiento para transferir la dirección contenida en `myPtr1` a `myPtr2`, esto vacía `myPtr1` y ahora `myPtr2` apunta hacia el entero 25.

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

Los argumentos para `std::make_unique<T>(...)` son exactamente los valores que de otro modo aparecerían en el inicializador entre llaves de una asignación dinámica de la forma `new T{...}`. En nuestro ejemplo, ese es el literal `double 999.0`. Para ahorrar algo de escritura, puede combinar esta sintaxis con el uso de la palabra clave `auto`:

```
auto pdata{ std::make_unique<double>(999.0) }
```

Puede desreferenciar `pdata` de la misma forma que lo haría con un puntero ordinario, y puede utilizar el resultado de la misma manera:

```
*pdata = 8888.0;  
std::cout << *pdata << std::endl; // Imprime 8888
```

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

La gran diferencia con el método tradicional de reservar memoria dinámica es que ya no hay que preocuparse por liberar la variable `double` del almacenamiento libre.

Puede acceder a la dirección que contiene un puntero inteligente llamando a su función `get()`.

```
std::cout << pdata.get() << std::endl;
```

Si queremos ver la dirección en hexadecimal:

```
std::cout << std::hex << std::showbase << pdata.get() << std::endl  
      << std::dec << std::noshowbase;
```

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

Todos los punteros inteligentes tienen una función `get()` que devolverá la dirección que contiene el puntero. Solo debe acceder al puntero crudo dentro de un puntero inteligente para pasarlo a funciones que usan este puntero solo brevemente, nunca a funciones u objetos que harían y conservarían una copia de este puntero. No se recomienda almacenar punteros crudos que apunten al mismo objeto que un puntero inteligente porque esto puede generar punteros colgantes nuevamente, así como todo tipo de problemas relacionados.

También puede crear un puntero único (`unique_ptr`) que apunte a una matriz.

Mostramos como hacerlo con la vieja sintaxis:

```
const size_t n {100}; // Tamaño array
std::unique_ptr<double[]> pvalues {new double[n]}; // Crea un array de n
// elementos
// dinamicamente
```

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

La forma antigua se muestra dado que seguramente encontrará código ya escrito que la utilice. Como siempre, recomendamos utilizar `std::make_unique<T[]>()` en lugar de la forma antigua.

```
auto pvalues{ std::make_unique<double[]>(n) }; // Crear array de n elementos  
                                                // dinámicamente
```

Para utilizar los elementos del array `pvalues`, utilizamos una notación similar a la que utilizamos con los punteros crudos:

```
for (size_t i {} ; i < n; ++i)  
    pvalues[i] = static_cast<double>(i + 1);
```

Este código establece los valores de los elementos del array de 1 a n.

# Smart pointers (punteros inteligentes)

Usando punteros `unique_ptr<T>`

El siguiente código imprime los elementos del array `pvalues`. Imprime 10 elementos por línea

```
for (size_t i {}; i < n; ++i)
{
    std::cout << pvalues[i] << ' ';
    if ((i + 1) % 10 == 0)
        std::cout << std::endl;
}
```

# Smart pointers (punteros inteligentes)

Usando punteros `unique_ptr<T>`

**RECOMENDACIÓN:** Siempre es recomendable utilizar el contenedor `vector<>` en vez de `unique_ptr<>`. Este es mucho más versátil y poderoso. Se presentan ejemplos simples con arrays para simplificar la explicación de este tipo de punteros. Su verdadero potencial se verá cuando veamos clases y objetos.

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

Puede restablecer el puntero contenido en un `unique_ptr<>`, o cualquier tipo de puntero inteligente, llamando a su función `reset()`:

```
pvalues.reset(); // La dirección es nullptr
```

`pvalues` todavía existe, pero ya no apunta a nada. Este es un objeto `unique_ptr<double>`, por lo que debido a que no puede haber otro puntero único que contenga la dirección del array, la memoria del array se liberará como resultado. Naturalmente, puede verificar si un puntero inteligente contiene `nullptr` comparándolo explícitamente con `nullptr`, pero un puntero inteligente también se convierte convenientemente en un valor booleano de la misma manera que un puntero crudo (es decir, se convierte en falso si y solo si contiene `nullptr`):

```
if (pvalues) // Equivale a: if (pvalues != nullptr)  
    std::cout << "El primer elemento es: " << pvalues[0] << std::endl;
```

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

Puede crear un puntero inteligente que contenga `nullptr` usando llaves vacías, `{}`, o simplemente omitiendo las llaves:

```
std::unique_ptr<int> my_number; // equivale a: ... my_number{};  
                                // equivale a: ... my_number{ nullptr };  
if (!my_number)  
    std::cout << "my_number no apunta a nada aun" << std::endl;
```

Crear punteros inteligentes vacíos sería de poca utilidad, si no fuera porque siempre puede cambiar el valor al que apunta un puntero inteligente. Puedes hacer esto de nuevo usando `reset()`:

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

```
my_number.reset(new int{ 123 }); // my_number apunta a un entero 123  
my_number.reset(new int{ 42 }); // my_number apunta a un entero 42
```

Llamar a `reset()` sin argumentos es equivalente a llamar a `reset(nullptr)`. Al llamar a `reset()` en un objeto `unique_ptr<T>`, con o sin argumentos, se desasignará cualquier memoria que haya pertenecido previamente a ese puntero inteligente. Entonces, con la segunda declaración en el fragmento anterior, la memoria que contiene el valor entero 123 se desasigna, después de lo cual el puntero inteligente toma posesión de la posición de memoria que contiene el número 42.

Junto a `get()` y `reset()`, un objeto `unique_ptr<T>` también tiene una función miembro llamada `release()`. Esta función se utiliza esencialmente para convertir el puntero inteligente en un puntero crudo.

# Smart pointers (punteros inteligentes)

Usando punteros `unique_ptr<T>` - Release

```
auto pvalues{ std::make_unique<double[]>(n) }; // Crear array de n elementos  
                                                // dinamicamente
```

A través del método `release`, `unique_ptr` renuncia a la propiedad del puntero a la zona de memoria reservada (lo libera) y devuelve un puntero crudo que apunta a dicha zona reservada. Este puntero crudo devuelto debe ser almacenado dado que ahora es responsabilidad nuestra liberar la zona de memoria reservada.

```
double *ptr_rls = pvalues.release();  
...  
...  
delete[] ptr_rls;  
ptr_rls = nullptr;
```

# Smart pointers (punteros inteligentes)

Usando punteros `unique_ptr<T>` - Release

Ejemplos de uso de release:

Puede haber instancias en las que ya no desee que `unique_ptr` sea propietario de los datos administrados y, sin embargo, no destruya el objeto; tal vez desee llamar a una API heredada que toma un puntero simple y asume su propiedad. O tal vez desee que su interfaz reciba un `unique_ptr` pero que tenga muchos `shared_ptrs` con acceso a él en la implementación. Por lo tanto, la implementación se liberaría de `unique_ptr` y transferiría la propiedad a uno o más `shared_ptrs`.

# Smart pointers (punteros inteligentes)

## Usando punteros `unique_ptr<T>`

**CUIDADO:** nunca llame a `release()` sin capturar el puntero crudo que devuelve. Es decir, nunca escriba una declaración de la siguiente forma:

```
pvalues.release();
```

¿Por qué? ¡Porque esto introduce una gran fuga de memoria, por supuesto! Libera (con `release()`) el puntero inteligente de la responsabilidad de desasignar la memoria, pero como no captura el puntero crudo, no hay forma de que usted o cualquier otra persona pueda aplicarle `delete` o `delete[]` nunca más. Si bien esto puede parecer obvio ahora, se sorprendería de la frecuencia con la que se llama por error a `release()` cuando, en cambio, se pretendía una declaración `reset()` de la siguiente forma:

```
pvalues.reset(); // No es lo mismo que release(); !!!
```

# Smart pointers (punteros inteligentes)

Usando punteros `shared_ptr<T>`

Declaración de un objeto `shared_ptr<T>`

```
std::shared_ptr<double> pdata {new double{999.0}};
```

Lo podemos desreferenciar de la misma manera que un puntero crudo o un objeto `unique_ptr<t>`:

```
*pdata = 8888.0;  
std::cout << *pdata << std::endl; // imprime 8888  
*pdata = 8889.0;  
std::cout << *pdata << std::endl; // imprime 8889
```

# Smart pointers (punteros inteligentes)

## Usando punteros `shared_ptr<T>`

Un objeto `shared_ptr<T>` también se comporta como un puntero al tipo `T`, pero a diferencia de `unique_ptr<T>`, puede haber cualquier número de objetos `shared_ptr<T>` que contengan, o compartan, la misma dirección. Por lo tanto, los objetos `shared_ptr<>` permiten la propiedad compartida de un objeto en el almacenamiento libre. En cualquier momento dado, en tiempo de ejecución de conoce el número de objetos `shared_ptr<>` que contienen una dirección dada en el tiempo. Esto se llama **conteo de referencias**. El recuento de referencias para `shared_ptr<>` que contiene una dirección de almacenamiento libre dada se incrementa cada vez que se crea un nuevo objeto `shared_ptr<>` que contiene esa dirección, y disminuye cuando se destruye un `shared_ptr<>` que contiene la dirección o se asigna para que apunte a una dirección diferente.

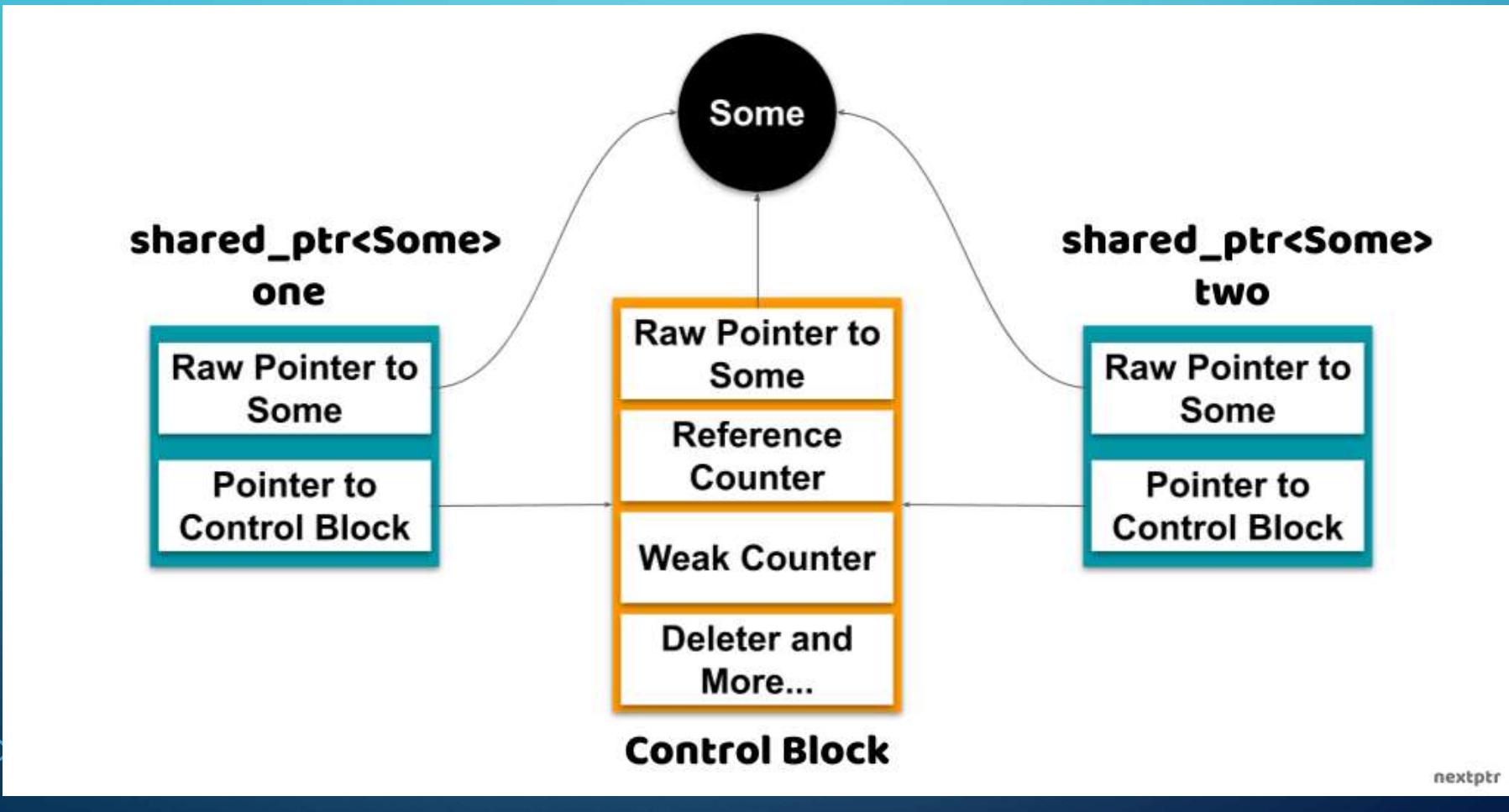
# Smart pointers (punteros inteligentes)

## Usando punteros `shared_ptr<T>`

Cuando no hay objetos `shared_ptr<>` que contengan una dirección determinada, el recuento de referencias se habrá reducido a cero y la memoria para el objeto en esa dirección se liberará automáticamente. Todos los objetos `shared_ptr<>` que apuntan a la misma dirección tienen acceso al recuento de cuántos hay.

# Smart pointers (punteros inteligentes)

Punteros `shared_ptr<T>`



# Smart pointers (punteros inteligentes)

## Usando punteros `shared_ptr<T>`

La creación de un objeto `shared_ptr<T>` implica un proceso más complicado que la creación de un objeto `unique_ptr<T>`, sobre todo por la necesidad de mantener un recuento de referencias. La definición de `pdata` implica una asignación de memoria para la variable `double` y otra asignación relacionada con el objeto de puntero inteligente. La asignación de memoria en el almacenamiento libre es costosa a tiempo. Puede hacer que el proceso sea más eficiente utilizando la función `make_shared<T>()` que se define en el encabezado `memory` para crear un puntero inteligente de tipo `shared_ptr<T>`:

```
auto pdata{ std::make_shared<double>(999.0) }; // Apunta a una variable  
                                                // tipo double
```

Esta declaración asigna memoria para la variable `double` y asigna memoria para el puntero inteligente en un solo paso, por lo que es más rápido. El tipo inferido por `auto` es `shared_ptr<double>`.

# Smart pointers (punteros inteligentes)

## Usando punteros `shared_ptr<T>`

Puede inicializar un `shared_ptr<T>` con otro `shared_ptr<T>` cuando lo define:

```
std::shared_ptr<double> pdata2 {pdata};
```

`pdata2` apunta a la misma variable que `pdata`. También puede asignar un `shared_ptr<T>` a otro `shared_ptr<T>` :

```
std::shared_ptr<double> pdata{new double {999.0}};
std::shared_ptr<double> pdata2; // Puntero contiene nullptr
pdata2 = pdata; // Copia puntero - ambos apuntan a la misma variable
std::cout << *pdata2 << std::endl; // Imprime 999
```

Copiar `pdata` aumenta el contador de referencias. Ambos punteros tienen que ser reiniciados o destruidos para la memoria ocupada por la variable `double` sea liberada.

# Smart pointers (punteros inteligentes)

Usando punteros `shared_ptr<T>`

Otra opción, por supuesto, es almacenar la dirección de un objeto contenedor `array<T>` o `vector<T>` que cree en el almacenamiento libre. Ejemplo:

# Smart pointers (punteros inteligentes)

## Usando punteros `shared_ptr<T>`

En la carpeta perteneciente a semana 2 en la subcarpeta Ejemplos (Ejemplo 2.1) , puede ver un ejemplo de utilización de `shared_ptr` que almacena la dirección de de un vector creado en el almacenamiento libre. En dicho ejemplo se calcula la temperatura promedio de un conjunto de muestras ingresadas por teclado.

*Es evidente que en este ejemplo citado hubiese sido mucha mas sencillo y práctico utilizar directamente un contenedor vector. Dicho ejemplo nos permite ilustrar el uso de este tipo de punteros y resaltar el hecho que puede ser utilizado con cualquier tipo de datos, incluidos los tipos creados por los usuarios (clases).*

# Smart pointers (punteros inteligentes)

## Ejercicio punteros unique\_ptr<T>

Realice un programa que reserve dinámicamente memoria utilizando `unique_ptr<T>`. El mismo deberá calcular la mediana de una muestra de datos ingresados por teclado. Imprima los elementos de la muestra y el valor de la mediana. El valor de la mediana para una cantidad de datos impar, es el valor central de los datos ordenados. Si la cantidad de datos es par es la media de los valores centrales. Ayuda:

La biblioteca estándar (std) proporciona el método `sort()`. Debe incluir el header `algorithm` para poder utilizarla. **Ejemplo:**

```
double data[5] { 7.0, 1.0, 2.0, 6.5, 10.1};  
  
sort(data, data + 5);
```

# Funciones

Forma general de una función en C++:

```
tipo_retorno nombre_funcion(lista_de_parametros)
{
    // Bloque de instrucciones....
}
```

Cuando se invoca una función con argumentos que no son del mismo tipo que los parámetros de la definición de la función, el compilador tratará de realizar una conversión implícita al tipo esperado. Si la conversión es de ampliación (promoción), el compilador no emitirá ninguna alerta, mientras que si es una conversión de reducción (coerción) posiblemente se emitirá una alerta de conversión de estrechamiento. Si la conversión no es posible se emitirá un error.

La combinación del nombre de la función mas la lista de parámetros se conoce como *firma de una función*.

# Funciones

Ejemplo:

```
double power(double num, int exp)
{
    double result {1.0};

    if(exp >= 0)
    {
        for(int n{1}; n <= exp; ++n)
        {
            result *= n;
        }
    }
    else
    {
        for (int n {1}; n <= -exp; ++n)
            result /= num;
    }
    return result;
}
```

```
double number {3.0};
const double result { power(number, 2) };
```

Definición

Uso

# Funciones

**void**

La palabra clave **void** indica que la función no retorna valores.

```
void printListTemp(double *temp)
{
    ....
}
```

Las funciones void pueden utilizar la palabra **return** para retornar desde cualquier punto de la función, obviamente sin un valor de retorno, es decir:

**return;**

Todas las variables definidas dentro de la función son de tipo **automáticas**, salvo las que son definidas como static.

# Funciones

## Retorno de valores

Las funciones no void que retornan valores pueden retornar cualquier tipo de objeto (tipos fundamentales, contenedores (vector<T>, array<T>), array, tipos definidos por el usuario. Forma general:

`return expresion;`

```
double calcAvgTemp(double *temp)
{
    ....
    return (valor_retorno);
}
```

# Funciones

## Prototipo de funciones

Suponga el siguiente fragmento:

```
int main()
{
    // Calcular potencias de 8 desde -3 a +3
    for (int i {-3}; i <= 3; ++i)
        std::cout << std::setw(10) << power(8.0, i);

    std::cout << std::endl;
}

double power(double num, int exp)
{
    double result {1.0};

    if(exp >= 0)
    {
        for(int n{1}; n <= exp; ++n)
        {
            result *= n;
        }
    }
    else
    {
        for (int n {1}; n <= -exp; ++n)
            result /= num;
    }
    return result;
}
```

# Funciones

## Prototipo de funciones

Este programa no compilará correctamente, el compilador le dirá que la función power no fue declarada y sin embargo lo esta. Esto es porque el compilador analiza el código fuente desde en orden desde el comienzo del mismo hacia el final (top-down) y cuando trata de encontrar la definición de power en main la misma todavía no ha sido definida.

La solución para esto es el *prototipo de funciones*.

```
double power(double x, int n); //Declaramos la función power
```

Si colocamos esta línea antes de la definición de main, el programa anterior compilara sin problemas. Al procesar esta sentencia el compilador buscará la definición de power dentro del archivo que contiene a main y en todos los archivos de cabecera que hayamos incluido en nuestro código fuente.

# Funciones

## Pasar argumentos a una función

Hay dos mecanismos por los cuales los argumentos se pasan a las funciones:

- Por valor
- Por referencia

Dado que estos mecanismos son conocidos, haremos hincapié en **paso por valor de punteros**.

Los punteros son pasados por valor. **Ejemplo:**

```
void accumulate(long *acum, int val)
{
    *acum += val;
}

int main()
{
    long acum{};

    accumulate(&acum, 5);
    accumulate(&acum, 25);
    accumulate(&acum, -7);

    cout << "Acumulado hasta el momento: " << acum << endl;
```

# Funciones

## Pasar argumentos a una función - Arrays

puede pasar la dirección de un array a una función simplemente usando su nombre. La dirección del array se copia y se pasa a la función. El paso del segundo elemento(size) es esencial, aplicar el operador `sizeof` o `std::size` sobre `array_v` no sirve, ya que `array_v` es un puntero, no un array. El uso de `std::size` con un puntero ocasiona un error.

```
/*
Demostración de acceso a elementos del array a través
de []. El parámetro de función es un puntero a array tipo
long.
*/
void ptr_array_demo1_funct(long *array_v, size_t size){
    for(size_t ind{}; ind < size; ++ind){
        cout << array_v[ind];
        cout << (ind < (size - 1)?"," ":"");
    }
    cout << endl;
```

# Funciones

## Pasar argumentos a una función - Arrays

Ejemplo de uso:

```
int main() {  
    long demo_array[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};  
    ptr_array_demo1_funct(demo_array, std::size(demo_array));  
}
```

La siguiente función también recibe un puntero a un array, no se deje engañar por la notación `array_v[]`.

# Funciones

## Pasar argumentos a una función – Arrays

```
/*
Demostración de acceso a elementos del array a través
de []. El parámetro de función es un puntero a array tipo
long.
*/
void ptr_array_demo1_funct(long array_v[], size_t size){

    for(size_t ind{}; ind < size; ++ind){

        cout << array_v[ind];
        cout << (ind < (size - 1)? "," : "");
    }
    cout << endl;
}
```

Realmente no hay ninguna diferencia en la forma en que se evalúan estas definiciones de funciones. De hecho, el compilador considera que los siguientes dos prototipos de funciones son idénticos:

# Funciones

## Pasar argumentos a una función – Arrays

```
void ptr_array_demo1_funct(long *array_v, size_t size);
void ptr_array_demo1_funct(long array_v[], size_t size);
```

Tenga cuidado si encuentra una definición de función similar a la siguiente:

```
double average10(double array[10]) /* El [10] no significa lo que podría esperar! */
{
    double sum {};//Acumula total

    for (size_t i {}; i < 10; ++i)
        sum += array[i]; // Suma elementos del array

    return sum / 10; // Retuna valor medio.
}
```

Es claro que la intención del programador fue indicar que esta función calcula el promedio de un array de 10 valores, pero en la práctica esta función puede recibir un puntero a un array de cualquier dimensión sin problemas.

# Funciones

## Pasar argumentos a una función – Arrays

Si por ejemplo usted invoca a la función anterior así:

```
double temps[] {25,32,52,45};  
average10(temps);
```

La función `average10(..)` calculará sobre 10 valores el promedio, dado que el array `temps` solo tiene 5 elementos los 5 restantes los tomará mas allá de los límites del array, calculando un valor no válido. Imagínese si la función escribiera sobre los valores del array, podría originar un excepción en el mejor de los casos. Si el array pasado como argumento fuese de mas de 10 elementos simplemente calcular un valor erróneo para el promedio. En síntesis: `funcion(tipo array[n])` equivale a `funcion(tipo array[])` y equivale a `funcion(tipo *array)`.

# Funciones

## Pasar argumentos a una función – Arrays

Si por ejemplo usted invoca a la función anterior así:

```
double temps[] {25,32,52,45};  
average10(temps);
```

La función `average10(..)` calculará sobre 10 valores el promedio, dado que el array `temps` solo tiene 5 elementos los 5 restantes los tomará mas allá de los límites del array, calculando un valor no válido. Imagínese si la función escribiera sobre los valores del array, podría originar un excepción en el mejor de los casos. Si el array pasado como argumento fuese de mas de 10 elementos simplemente calcular un valor erróneo para el promedio. En síntesis: `funcion(tipo array[n])` equivale a `funcion(tipo array[])` y equivale a `funcion(tipo *array)`.

# Funciones

## Pasar argumentos a una función – Parámetros de puntero constante

Si declaramos el parámetro de la función como puntero constante, no podemos modificar el valor apuntado, en el siguiente ejemplo no podemos modificar los elementos de array\_v. Esto se utiliza cuando diseñamos funciones de solo lectura. Esto aplica también a aquellos objetos que no son arrays (tipos fundamentales, contenedores, clases, etc) que se pasan como parámetros a través de punteros.

```
/*
Demostración de acceso a elementos del array a través
de []. El parámetro de función es un puntero a array tipo
long.
*/
void ptr_array_demo1_funct(long array_v[], size_t size){

    ...
    array_v[4] = 20;      //Error! No podemos modificar array.
    ...
}
```

# Funciones

## Pasar argumentos a una función – Parámetros por referencia

Como hemos visto las referencias actúan como alias de un variable u objeto. Una vez que asignamos la referencia, esta apunta hacia la variable u objeto de igual manera que un puntero. La ventaja sobre los punteros es que no necesitamos desreferenciar para leer o asignar valores a la referencia. Las referencias se inicializan cuando se definen y no pueden ser reasignadas una vez establecidas:

```
int val {258};  
int& ref_val {val};  
ref_val = 325;
```

Así como los punteros, podemos utilizarlas como parámetros de una función, siendo su uso mas sencillo dado que las tratamos como si fueran parámetros pasados por valor (no hay desreferenciar).

# Funciones

Pasar argumentos a una función – Parámetros por referencia

Ejemplo:

```
void change_par_val_by_ref(int& value)
{
    value = 250;
}

int main()
{
    int main_val;
    change_par_val_by_ref(main_val);
    cout << main_val << endl; // Imprime 250.
}
```

main\_val es pasado por referencia a la función change\_par\_val\_by\_ref, observe que dentro de la función el parámetro value es tratado de la misma manera que es tratado un parámetro pasado por valor. Luego de ejecutar la función main\_val asume el valor 250. Si el parámetro value es declarado como constante, ya no podremos alterar el valor del mismo.

# Funciones

Pasar argumentos a una función – Parámetros por referencia

Ejemplo con referencia constante:

```
void change_par_val_by_ref(const int& value)
{
    value = 250;      // Error!! La referencia es const
}
int main()
{
    int main_val;
    change_par_val_by_ref(main_val);
    cout << main_val << endl;
}
```

Al igual que con los punteros, declarar un parámetro pasado por referencia como constante impide la modificación de dicho parámetro y por ende de la variable u objeto asociado a la referencia.

# Funciones

## Pasar argumentos a una función – Parámetros por referencia

Si la función que diseña solo necesita acceder a los valores pero no alterarlos utilice siempre referencias constantes como argumento de dicha función. Debido a que la función no cambiará un parámetro de referencia a `const`, el compilador permitirá tanto argumentos `const` como `nonconst`.

Pero solo se pueden proporcionar argumentos no constantes para un parámetro de referencia a no constante.

Llamar a una función que tiene un parámetro de referencia es sintácticamente indistinguible de llamar a una función donde el argumento se pasa por valor. Alguien que no vea la firma de la función puede pensar que esta pasando el argumento por valor.

# Funciones

## Pasar argumentos a una función – Pasar arrays por referencia

Aquí se muestra la forma de pasar un array por referencia.

Observe que se debe indicar el tamaño exacto del array que se pasará. Esta es una versión funcional del ejemplo con punteros

```
double average10(const double (&array)[10]) // Solamente se puede pasar
                                              // arrays de 10 elementos!
{
    double sum {};// Para almacenar la suma de los elementos del array

    for (size_t i {}; i < 10; ++i)
        sum += array[i]; // Suma los elementos del array

    return sum / 10; // Calcula promedio.
}
```

Pasar el array por referencia permite utilizar el operador `sizeof`, la función `std::size` y `for` basado en rangos. Dada la existencia del contenedor `std::array<T>`, no tiene mucho sentido pasar un array por referencia.

# Funciones

Pasar argumentos a una función – Pasar arrays por referencia

Ejemplo con uso de función std::size:

```
double average10(const double (&array)[10]) // Solamente se puede pasar
                                              // arrays de 10 elementos!
{
    double sum {}; // Para almacenar la suma de los elementos del array

    for (size_t i {}; i < 10; ++i)
        sum += array[i]; // Suma los elementos del array

    return sum / std::size(array); // Calcula promedio.
}
```

Forma moderna de realizar la función anterior

```
double average10(const std::array<double,10>& values)
{
    double sum {}; // Para almacenar la suma de los
                  // elementos del array.
    for (auto elem : values)
        sum += elem; // Suma los elementos del array

    return sum / std::size(values); // Calcula promedio.
}
```

# Funciones

## Pasar argumentos a una función – Referencias y conversiones implícitas

Supongamos que tenemos el siguiente código:

```
// Conversiones implicitas de parámetros
// pasados por referencia.
#include <iostream>

void double_it(double& it)
{
    it *= 2;
}

void print_it(const double& it)
{
    std::cout << it << std::endl;
}

int main()
{
    double d{123};

    double_it(d);
    print_it(d);

    int i{456};
    double_it(i); /* error, no compila! */
    print_it(i);
}
```

# Funciones

## Pasar argumentos a una función – Referencias y conversiones implícitas

La primera parte del código funciona acorde a lo esperado ya que pasamos como argumento de la función `double_it` un valor de tipo `double` (`d`). Lo interesante esta en las últimas líneas. Analicemos la función `print_it()` que espera como parámetro una referencia a `double`. Pero el valor que le pasamos a la función como argumento no es doble; es un `int`! Este `int` generalmente tiene solo 4 bytes de tamaño, y sus 32 bits están dispuestos de manera completamente diferente a los de un doble. Entonces, ¿cómo puede esta función leer desde un alias para un doble si no hay tal doble definido en ninguna parte del programa? La respuesta es que el compilador, antes de llamar a `print_it()`, crea implícitamente un valor doble temporal en algún lugar de la memoria, le asigna el valor `int` convertido y luego pasa una referencia a esta ubicación de memoria temporal a `print_it()`.

# Funciones

## Pasar argumentos a una función – Referencias y conversiones implícitas

Estas conversiones implícitas solo se admiten para parámetros de referencia a const, no para parámetros de referencia a no constante. En el caso de `double_it(i)`, la referencia no es const por tanto se intentará escribir la variable `i` luego del cálculo, pero esta es tipo int. Para que esto funcionara deberían ocurrir 2 procesos:

1. Se crea implícitamente un valor double temporal en algún lugar de la memoria, le asigna el valor int convertido y luego pasa una referencia a esta ubicación de memoria temporal a la función `double_it()` y le aplicará el cuerpo de función de `double_it()`.

# Funciones

## Pasar argumentos a una función – Referencias y conversiones implícitas

2. Entonces tendría un doble temporal en alguna parte, ahora con valor 912.0, y un valor `int i` que sigue siendo igual a 456. Ahora, mientras que en teoría el compilador podría convertir el valor temporal resultante de nuevo en un `int`, los diseñadores de C++ decidieron que esto no es válido. La razón es que, por lo general, tales conversiones inversas significarían inevitablemente la pérdida de información. En nuestro caso, esto implicaría una conversión de `double` a `int`, lo que resultaría en la pérdida de al menos la parte fraccionaria del número. Por lo tanto, nunca se permite la creación de temporales para parámetros de referencia a no constantes. Esta es también la razón por la cual la instrucción `double_it(i)` siendo `i` un tipo de menor tamaño que el especificado en la definición de la función, no es válida en C++ estándar y debería fallar al compilarse.

# Funciones

## Pasar argumentos a una función – Valores de argumento por defecto

C++ permite establecer parámetros con valores defecto. Puede especificar el / los parámetros por defecto en el prototipo o en la definición de la función, si lo hace en el prototipo no lo puede repetir en la definición. **Ejemplo: Especificado en la definición:**

```
double calculo_porcentual(unsigned int valor, unsigned porcentaje = 70)
{
    return(valor * porcentaje / 100.0);
}
```

Parámetro por defecto proporcionado en el prototipo, si lo repite en la definición se produce un error de compilación:

```
double calculo_porcentual(unsigned int valor, unsigned porcentaje = 70);

double calculo_porcentual(unsigned int valor, unsigned porcentaje)
{
    return(valor * porcentaje / 100.0);
}
```

# Funciones

Pasar argumentos a una función – Valores de argumento por defecto

Ejemplos de uso:

```
cout << calculo_porcentual(583) << endl;// calcula e imprime el 70% de 583.
```

```
cout << calculo_porcentual(583, 55) << endl;// calcula el 55% de 583.
```

Múltiples parámetros por defecto

```
unsigned long dummy(const unsigned int data[], size_t data_size = 1, bool isSuma = true)
{
    unsigned long suma {};

    for(size_t n{}; n < data_size; ++n)
    {
        cout << data[n] << ((n+1) < data_size ? ", " : "");
        isSuma ? suma += data[n] : 0;
    }

    cout << endl;
    if(isSuma) cout << "Suma de los elementos: " << suma << endl;

    return suma;
}
```

# Funciones

## Pasar argumentos a una función – Valores de argumento por defecto

Vale la misma aclaración que hicimos con el ejemplo con un solo parámetro:

Puede especificar el / los parámetros por defecto en el prototipo o en la definición de la función, si lo hace en el prototipo no lo puede repetir en la definición. Puede omitir argumentos solo al final de la lista; no se le permite omitir el segundo en el fragmento de código anterior por ejemplo.

### Ejemplo de uso:

```
unsigned int dato{99};  
imprime_suma(&dato);  
  
array<unsigned int, 20> valores;  
valores.fill(255);  
  
cout << imprime_suma(valores.data(), valores.size()) << endl; // calcula e imprime  
// suma de elementos.
```

# Funciones

## Retornando valores de una función – Retornando un puntero

Cuando devuelve un puntero de una función, debe contener nullptr o una dirección que aún sea válida en la función de llamada. En otras palabras, la variable a la que se apunta aún debe estar dentro del alcance después del regreso a la función de llamada. Esto implica la siguiente regla absoluta:

*Nunca devuelva la dirección de una variable local automática desde una función.*

# Funciones

## Retornando valores de una función – Retornando un puntero

Vamos a trabajar con un ejercicio que deberá completar y en el proceso aplicaremos devolución de punteros. Vamos a implementar una normalización de valores de un vector. Este tipo de proceso se utiliza en estadística y machine learning. Vamos a partir explicando como se normaliza una muestra de valores. El primer paso es encontrar el valor mínimo de la muestra, luego el valor máximo y por último calcular el valor normalizado. Para normalizar tomamos cada elemento de la muestra, le restamos el mínimo (Con esto hacemos que el conjunto de muestras se encuentre en el rango de 0 a 1) y luego dividimos por la diferencia entre el máximo y el mínimo del conjunto de muestras.

$$z_i = (x_i - \text{mínimo}(x)) / (\text{máximo}(x) - \text{mínimo}(x))$$

# Funciones

## Retornando valores de una función – Retornando un puntero

Las siguientes funciones podemos observar la devolución de valores por punteros. Observe que lo que se devuelve es la dirección del elemento del array que representa el mínimo/máximo de la muestra. En ningún momento se devuelve la dirección de una variable local.

Estas funciones son parte del paquete de funciones que se necesita para completar nuestro ejemplo de normalización. Usted podría codificar las restantes?. Tenga en cuenta que restan:

1. Una función para convertir la muestra en valores solo positivos (`shift_samples(...)`)
2. Una función que convierta cada muestra en un valor de rango 0.0 a 1.0
3. Una función que imprima las muestras normalizadas.

# Funciones

# Funciones

# Funciones

## Retornando valores de una función – Retornando un puntero

El hecho de retornar un puntero en las funciones `find_min` y `find_max` es a modo de ilustración. En realidad hubiese sido mucho mas simple devolver el índice del elemento que representa el máximo, pero así como esta diseñada sirve a nuestro propósito

# Funciones

## Retornando valores de una función – Retornando referencias

Devolver un puntero desde una función es útil, pero puede ser problemático. Los punteros pueden ser nulos, y la desreferenciación de `nullptr` generalmente resulta en la falla de su programa. La solución, es devolver una referencia. Dado que una referencia es un alias para otra variable, por lo que podemos establecer la siguiente regla de oro para las referencias:

*Nunca devuelva una referencia a una variable local automática en una función*

# Funciones

## Retornando valores de una función – Retornando referencias

Ejemplo: Encontrar el mayor de 2 strings.

```
string& mayor(string& s1, string& s2)
{
    return s1 > s2? s1 : s2; // Retorna una referencia al string mayor
}
```

Al devolver una referencia, se puede utilizar la llamada a la función (`mayor(..)` en este caso) a la izquierda de una asignación sin necesidad de desreferenciar. Obviamente que también puede ser utilizada a la derecha de una asignación.

Ejemplo:

```
string s1 {"Jose"}, s2{"Carlos"};
string s3 {mayor(s1, s2)};
mayor(s1, s2) = "Marcelo";
```

# Funciones

## Retornando valores de una función – Retornando referencias

Debido a que los parámetros no son constantes, no puede usar cadenas literales como argumentos; el compilador no lo permitirá. Un parámetro por referencia permite que se cambie el valor, y cambiar una constante no es algo que el compilador aceptará a sabiendas. Si hace que los parámetros sean constantes, no puede usar una referencia a no constante como tipo de devolución.

# Funciones

## Retornando valores de una función – Retornando referencias

**RECOMENDACIÓN:** En C++ moderno, generalmente debería preferir devolver valores sobre parámetros de salida. Esto hace que las firmas de funciones y las llamadas sean mucho más fáciles de leer. los argumentos son para la entrada y se devuelve toda la salida. El mecanismo que hace esto posible se llama semántica de movimiento. En pocas palabras, la semántica de movimiento asegura que devolver objetos que administran la memoria asignada dinámicamente, como vectores y cadenas, ya no implica copiar esa memoria y, por lo tanto, es muy barato. las excepciones notables son las matrices u objetos que contienen un array, como `std::array<>`. para estos es aún mejor usar parámetros de salida.

# Funciones

## Retornando valores de una función – Deducción del tipo a retornar

Así como el compilador puede deducir el tipo de una variable a través de su inicialización, también puede deducir el tipo de retorno de una función en base al valor devuelto. Ejemplo:

```
auto getAnswer() { return 42; }
```

Es obvio que el tipo devuelto es int, inferido en base al valor 42. Este ejemplo es muy simple y quizás no ahorremos escritura si escribimos auto en vez de int. Cuando veamos tipos mas complejos cuyos tipos son mas verbosos, será de gran ayuda la inferencia de tipos.

# Funciones

Retornando valores de una función

Deducción del tipo a retornar y referencias

Se debe tener cuidado con la deducción de tipos cuando el tipo de retorno es una referencia: Por ejemplo, si utilizamos un ejemplo anterior:

```
auto mayor(string& s1, string& s2)
{
    return s1 > s2? s1 : s2; // Retorna una referencia al string mayor
}
```

El tipo inferido aquí no es `string&`, el tipo deducido es `string`. Por tanto será retornada una copia de `s1` o `s2`, no una referencia.

# Funciones

Retornando valores de una función

Deducción del tipo a retornar y referencias

Si lo que quiere es retornar una referencia en la función mayor(..) sus opciones serán:

- Especificar explícitamente el tipo de retorno `std::string&` como antes.
- Especificar `auto&` en lugar de `auto`. Entonces el tipo de devolución siempre será una referencia.

# Funciones

## Funciones inline

Existen funciones que son cortas y la sobrecarga que el compilador genera para pasar argumento y devolver resultados es significativa comparada con el tiempo de ejecución del cuerpo de la función. En tales circunstancias, puede sugerirle al compilador que reemplace una llamada de función con el código del cuerpo de la función. La palabra clave **inline** en la definición de la función nos permite hacer esto.

```
inline int find_max(long val1, long val2)
{
    return val1 > val2 ? val1 : val2;
}
```

# Funciones

## Funciones inline

Sin embargo, es solo una sugerencia, y depende del compilador si se acepta su sugerencia. Cuando una función se especifica como en línea, la definición debe estar disponible en cada archivo fuente que llama a la función. Por este motivo, la definición de una función en línea suele aparecer en un archivo de encabezado en lugar de en un archivo de origen, y el encabezado se incluye en cada archivo de origen que utiliza la función. La mayoría, si no todos los compiladores modernos, harán funciones cortas en línea, incluso cuando no usa la palabra clave en línea en la definición. Si no lo hace, recibirá mensajes "unresolved external" cuando se vincule el código.

# Funciones

## Sobrecarga de funciones

Las funciones sobrecargadas tienen el mismo nombre, por lo que la firma de cada función sobrecargada debe diferenciarse solo por la lista de parámetros. Eso permite que el compilador seleccione la función correcta para cada llamada de función según la lista de argumentos. Dos funciones con el mismo nombre son diferentes si al menos uno de los siguientes es verdadero:

- Las funciones tienen diferente número de parámetros.
- Al menos un parámetro es de tipo diferentes.

*El tipo de retorno no es parte de la firma de la función, por lo tanto no se utiliza para determinar a cual función debe llamarse.*

# Funciones

## Sobrecarga de funciones

¿Como se determina a que función invocar?:

1. Coincidencia exacta (Nombre de la función y parámetros). Ejemplo:

```
int funcion(int, char)
{
...
}

int a{7};
char b{25};

func(a, b);
```

2. No se encuentra coincidencia exacta:

char, unsigned char, y short son promovidos a int.  
float es promovido a double.

# Funciones

## Sobrecarga de funciones

2. No se encuentra coincidencia exacta. Ejemplo:

```
int funcion(int, char)
{
    cout << "(int, char)" << endl;
    return 0;
}

int funcion(long, char)
{
    cout << "(long, char)" << endl;
    return 0;
}

int funcion(float, char)
{
    cout << "(float, char)" << endl;
    return 0;
}

int b{45};
char c{25}, d{120};

funcion(c, d); //invoca a (int,char)
```

# Funciones

## Sobrecarga de funciones

¿Como se determina a que función invocar?:

3. No se encuentra ninguna coincidencia:  
C++ tratará de hallar una coincidencia a través una conversión estándar.
4. Si fallan todas las verificaciones anteriores, se produce un error de compilación.

# Funciones

## Sobrecarga de funciones

### Parámetros tipo puntero

Los siguientes prototipos declaran funciones sobrecargadas:

```
int largest(int* pValues, size_t count); // Prototipo a
```

```
int largest(float* pValues, size_t count); // Prototipo b
```

### Parámetros por referencia

```
void do_it(std::string str1); // No son distinguibles...
```

```
void do_it(std::string& str1); // ...por el tipo de argumento
```

```
std::string word {"Hello"};  
do_it(word); // A cual funcion se llama?..no se  
// puede determinar
```

# Funciones

## Sobrecarga de funciones

### Parámetros por referencia

Recuerde que para funciones con algún prototipo de tipo referencia no constante, el compilador no usará una dirección temporal para inicializar dicha referencia.

```
double mayor(double a, double b);
long& mayor(long& a, long& b);

int main()
{
    ...
    mayor(static_cast<long>(a_int), static_cast<long>(b_int));
    ...
}
```

En este caso se invoca a la función `double mayor(double a, double b)`, no a la que posee parámetros `long&`. Para que la función con parámetros `long&` sea invocada, ambos parámetros deberían ser `const long&`.

# Funciones

## Sobrecarga de funciones

### Parámetros constantes

Un parámetro `const` solo se distingue de un parámetro `no const` para referencias y punteros. Para un tipo fundamental como `int`, por ejemplo, `const int` es idéntico a `int`. Por lo tanto, los siguientes prototipos no son distinguibles:

```
long mayor(long a, long b);  
long mayor(const long a, const long b);
```

Cuando procesa el archivo para determinar a que función se desea llamar, el compilador obvia el atributo `const` para parámetros de tipos fundamentales.

# Funciones

## Sobrecarga de funciones

### Parámetros punteros constantes

```
long* mayor(long* a, long* b); // Prototipo 1
```

```
const long* mayor(const long* a, const long* b); // Prototipo 2
```

```
long* mayor(long* const a, long* const b); // Identico a prototipo 1  
const long* mayor(const long* const a, const long* const b); // Identico a  
// prototipo 2
```

```
const long num4 {1L};  
const long num5 {2L};  
const long num6 {*mayor(&num4, &num5)}; // se invoca a prototipo 2
```

```
long num1 {1L};  
long num2 {2L};  
long num3 {*mayor(&num1, &num2)}; // se invoca a prototipo 1
```

# Funciones

## Sobrecarga de funciones

### Ejercicio:

Dado el siguiente fragmento, construya un programa que incluya este y 2 funciones adicionales que reciban `vector<int>`, `vector<double>` que devuelvan el mayor valor del vector.

```
string find_max(const vector<string>& words)
{
    string max_word {words[0]};

    for (const auto& word : words)

        if (max_word < word) max_word = word;
    return max_word;
}
```

# Plantillas de funciones

Una plantilla de función en sí misma no es una definición de una función; es un modelo para definir toda una familia de funciones. Una plantilla de función es una definición de función paramétrica, donde una instancia de función particular es creada por uno o más valores de parámetro. El compilador usa una plantilla de función para generar una definición de función cuando es necesario. Si nunca es necesario, no se obtiene ningún código de la plantilla. Una definición de función que se genera a partir de una plantilla es una instancia o una instanciación de la plantilla. Los parámetros de una plantilla de función suelen ser tipos de datos, donde se puede generar una instancia para un valor de parámetro de tipo int, por ejemplo, y otra con un valor de parámetro de tipo string. Pero los parámetros no son necesariamente tipos. Pueden ser otras cosas como una dimensión, por ejemplo.

# Plantillas de funciones

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

La plantilla de función comienza con la palabra clave `template` para identificarla como tal. A esto le sigue un par de corchetes angulares que contienen una lista de uno o más parámetros de plantilla. En este caso, solo hay uno, el parámetro `T`. `T` se usa comúnmente como un nombre para un parámetro porque la mayoría de los parámetros son tipos, pero puede usar cualquier nombre que desee para un parámetro; nombres como `mi_tipo` o `Comparable` son igualmente válidos. Si hay varios parámetros, se pueden usar nombres más descriptivos. La palabra clave `typename` identifica que `T` es un tipo. Por lo tanto, `T` se denomina *parámetro de tipo de plantilla*. También puede usar palabra clave `class` aquí, pero preferimos `typename` porque el argumento de tipo también puede ser un tipo fundamental, no solo un tipo de clase.

# Plantillas de funciones

El resto de la definición es similar a una función normal. El compilador crea una instancia de la plantilla al reemplazar T en toda la definición con un tipo específico. El tipo asignado a un parámetro de tipo T durante la instanciación se denomina argumento de tipo de plantilla. Puede colocar la plantilla en un archivo fuente de la misma manera que una definición de función normal; también puede especificar un prototipo para una plantilla de función. En este caso, sería de la siguiente manera:

```
template<typename T> T larger(T a, T b); // Prototipo para una plantilla de  
// funcion.
```

# Plantillas de funciones

## Creación de instancias

- Cada vez que se utiliza la función se crea una nueva instancia.
- A partir del tipo de los argumentos, se determina que instancia crear.
- La instancia se crea una única vez (la primera que se deduce a partir del tipo de los parámetros), es decir si hay 2 funciones con el mismo tipo de parámetros se deduce de la primera función invocada.

# Plantillas de funciones

## Parámetros tipo plantilla

El nombre de un parámetro de tipo plantilla se puede usar en cualquier parte de la firma de la función, el tipo de valor devuelto y el cuerpo de la plantilla. Es un marcador de posición para un tipo y, por lo tanto, puede colocarse en cualquier contexto en el que normalmente colocaría el tipo concreto. Es decir, supongamos que T es un nombre de parámetro de plantilla; entonces puede usar T para construir tipos derivados como T&, const T&, T\*, etc. O puede usar T como argumento para una plantilla de clase, como por ejemplo en std::vector<T>. Podemos generar una versión “anti copias” de nuestra plantilla.

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

# Plantillas de funciones

## Argumentos de plantilla explícitos

El siguiente código fallará:

```
int small_int {11};

std::cout << "El mayor de " << small_int << " y 19.6 es "
<< find_max(small_int, 19.6) << std::endl;
```

Los argumentos son de diferente tipo, por tanto el compilador no podrá deducir correctamente a que tipo se refiere y no podrá instanciar la función correspondiente a esta llamada. Por supuesto que podría realizar una conversión explícita con `static_cast<>` para `small_int`.

# Plantillas de funciones

## Argumentos de plantilla explícitos

Podría definir la plantilla para permitir que los parámetros para `find_max()` sean de diferentes tipos, pero esto agrega una complicación que discutiremos más adelante, además: ¿cuál de los dos usa para el tipo de retorno? Por ahora, concentrémonos en cómo puede especificar el argumento para un parámetro de plantilla explícitamente cuando llama a la función. Esto le permite controlar qué versión de la función se utiliza. El compilador ya no deduce el tipo para reemplazar T; acepta lo que usted especifica. Puede resolver el problema de usar diferentes tipos de argumentos con `find_max()` con una instancia explícita de la plantilla:

```
std::cout << "El mayor de " << small_int << " y 19.6 es "  
<< find_max<double>(small_int, 19.6) << std::endl; // Imprime 19.6
```

# Plantillas de funciones

## Argumentos de plantilla explícitos

Esto genera una instancia con T como tipo double. Esto insertará una conversión de tipo implícita a tipo double para el primer argumento (small\_int). También podemos indicarle al compilador que genere la instancia como tipo int, lo cual como se dará cuenta implica una perdida de precisión que generalmente no es deseable, pero a efectos ilustrativos podemos tomarnos esa libertad. Algunos compiladores podrían emitir una advertencia.

```
std::cout << "El mayor de " << small_int << " y 19.6 es "
<< find_max<int>(small_int, 19.6) << std::endl; // Imprime 19
```

# Plantillas de funciones

## Especialización de plantilla de función

Supongamos el siguiente caso:

```
int small_int {19};  
int big_int {15239987};  
  
std::cout << "El mayor de " << big_int << " y " << small_int << " es "  
<< *find_max(&big_int, &small_int) << std::endl; // La salida podría ser 19!
```

El compilador instancia la plantilla con el parámetro como tipo `int*`. El prototipo de esta instancia es el siguiente:

```
int* find_max(int*, int*);
```

# Plantillas de funciones

## Especialización de plantilla de función

El valor de retorno es una dirección, y debe desreferenciar para obtener el resultado. Sin embargo, el resultado puede muy bien ser 19, lo cual es incorrecto. Esto se debe a que la comparación es entre direcciones pasadas como argumentos, no los valores en esas direcciones. El compilador es libre de reorganizar las ubicaciones de memoria de las variables locales, por lo que el resultado real puede variar entre los compiladores. Esto ilustra lo fácil que es crear errores ocultos usando plantillas. Debe tener especial cuidado al usar tipos de punteros como argumentos de plantilla. Pruebe cambiar de lugar la definición de `small_int` (colóquela debajo de `big_int`) y vea los resultados.

# Plantillas de funciones

## Especialización de plantilla de función

Puede definir una especialización de la plantilla para acomodar un argumento de plantilla que es un tipo puntero. Para un valor de parámetro específico o un conjunto de valores en el caso de una plantilla con varios parámetros, una especialización de plantilla define un comportamiento que es diferente de la plantilla estándar. La definición de una especialización de plantilla debe colocarse después de una declaración (prototipo) o definición de la plantilla original. Si pone una especialización primero, entonces el programa no se compilará.

# Plantillas de funciones

## Especialización de plantilla de función

En este ejemplo declaramos el prototipo, por tanto la especialización se coloca luego del prototipo o luego de la definición

```
template <typename T> T find_max(T a, T b);

// La especialización de plantilla debe
// colocar después del prototipo o declaración
// (en caso que no se provea prototipo)
// la plantilla original.

template <>
int* find_max<int*>(int* a, int* b)
{
    return *a > *b ? a : b;
}

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

```
template <typename T> T find_max(T a, T b);

// La especialización de plantilla debe
// colocar después del prototipo o declaración
// (en caso que no se provea prototipo)
// la plantilla original.

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

template <>
int* find_max<int*>(int* a, int* b)
{
    return *a > *b ? a : b;
}
```

# Plantillas de funciones

## Especialización de plantilla de función

En este ejemplo **NO** declaramos el prototipo, por tanto la especialización debe colocarse obligatoriamente luego de la definición.

```
// La especialización de plantilla debe
// colocar después del prototipo o declaración
// (en caso que no se provea prototipo)
// la plantilla original.

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

template <>
int* find_max<int*>(int* a, int* b)
{
    return *a > *b ? a : b;
}
```

# Plantillas de funciones

## Especialización de plantilla de función

La especialización también debe aparecer antes de su primer uso. La definición de una especialización comienza con la palabra clave `template`, pero se omite el parámetro, por lo que los corchetes angulares que siguen a la palabra clave están vacíos. Todavía debe definir el tipo de argumento para la especialización y colocarlo entre corchetes angulares inmediatamente después del nombre de la función de plantilla. La definición de una especialización de `find_max()` para el tipo `int*` es la siguiente:

```
template <>
int* find_max<int*>(int* a, int* b)
{
    return *a > *b ? a : b;
}
```

# Plantillas de funciones

## Plantillas de funciones y sobrecarga

Puede sobre cargar una plantilla de función definiendo otras funciones con el mismo nombre. Por lo tanto, puede definir "anulaciones" para casos específicos, que el compilador siempre usará con preferencia a una instancia de plantilla. Como siempre, cada función sobre cargada debe tener una firma única. Reconsideraremos la situación anterior en la que necesita sobre cargar la función `find_max()` para tomar argumentos de puntero. En lugar de usar una especialización de plantilla para `find_max()`, podría definir una función sobre cargada. El siguiente prototipo de función sobre cargada lo haría:

```
int* find_max(int* a, int* b); //Función sobre cargada de  
// la plantilla find_max
```

# Plantillas de funciones

## Plantillas de funciones y sobrecarga

Luego definimos la función sobrecargada:

```
int* find_max(int* a, int* b)
{
    return *a > *b ? a : b;
}
```

También es posible sobre cargar una plantilla existente con otra plantilla. Por ejemplo, podría definir una plantilla que sobre cargue la plantilla `find_max()` para encontrar el valor máximo contenido en un array:

```
template <typename T>
T find_max(const T data[], size_t count)
{
    T result {data[0]};

    for (size_t i {1}; i < count; ++i)
        if (data[i] > result) result = data[i];

    return result;
}
```

# Plantillas de funciones

## Plantillas de funciones y sobrecarga

Podría definir una nueva plantilla que sobrecargue la plantilla `find_max()` para encontrar el valor máximo contenido en un vector:

```
template <typename T>
T find_max(const vector<T>& data)
{
    T result {data[0]};

    for (auto& value : data)
        if (value > result) result = value;

    return result;
}
```

También podemos definir una plantilla para punteros en lugar de utilizar la función sobrecargada vista con anterioridad.

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

# Plantillas de funciones

## Plantillas de funciones con parámetros múltiples

Como recordará el siguiente fragmento presentaba errores al compilarse, debido a que no podía ser deducido correctamente a qué tipo correspondía el parámetro T. Lo solucionamos especificando explícitamente el tipo.

```
std::cout << "El mayor de " << small_int << " y 19.6 es "
<< find_max(small_int, 19.6) << std::endl;
```

También podemos resolverlo a través de una plantilla con argumentos múltiples. De esta forma el compilador podrá deducir mas de un tipo durante la instanciaación.

```
template <typename T1, typename T2>
??? find_max(T1 a, T2 b)
{
    return a > b ? a : b;
}
```

# Plantillas de funciones

## Plantillas de funciones con parámetros múltiples

¿Cuál será el valor de retorno? ¿T1?, ¿T2?

Una primera solución posible es agregar un argumento de tipo plantilla adicional para proporcionar una forma de controlar el tipo de devolución. Por ejemplo:

```
template <typename TReturn, typename TArg1, typename TArg2>
TReturn find_max(TArg1 a, TArg2 b)
{
    return a > b ? a : b;
}
```

También podemos resolverlo a través de una plantilla con argumentos múltiples. De esta forma el compilador podrá deducir mas de un tipo durante la instanciaión.

# Plantillas de funciones

## Plantillas de funciones con parámetros múltiples

Sin embargo, el compilador no puede deducir este tipo de retorno, TReturn. La deducción de argumentos de plantilla funciona sobre la base de los argumentos pasados solo en la lista de argumentos de la función. Por lo tanto, siempre debe especificar el argumento de la plantilla TReturn usted mismo. Sin embargo, el compilador puede deducir el tipo de los argumentos, por lo que puede especificar solo el tipo de retorno. En general, si especifica menos argumentos de plantilla que la cantidad de parámetros de plantilla, el compilador deducirá los demás. Por lo tanto, las siguientes tres líneas son equivalentes:

```
std::cout << "El mayor de 1.5 y 2 es " << find_max<size_t>(1.5, 2) << std::endl;
std::cout << "El mayor de 1.5 y 2 es " << find_max<size_t, double>(1.5, 2) << std::endl;
std::cout << "El mayor de 1.5 y 2 es " << find_max<size_t, double, int>(1.5, 2) <<
std::endl;
```

# Plantillas de funciones

## Plantillas de funciones con parámetros múltiples

Claramente, la secuencia de parámetros en la definición de plantilla es importante aquí. Si tuviera el tipo de devolución como segundo parámetro, siempre tendría que especificar ambos parámetros en una llamada. Si especifica solo un parámetro, se interpretará como el tipo de argumento, dejando el tipo de retorno sin definir (tipo de retorno esta como segundo parámetro). Debido a que especificamos el tipo de devolución como `size_t`, en los tres casos, los resultados de estas llamadas a funciones serán todos 2. El compilador crea una función que acepta argumentos de tipo `double` e `int` y luego convierte su resultado en un valor de tipo `size_t`.

# Plantillas de funciones

## Plantillas de funciones con parámetros múltiples

Aún con esto todavía no podemos resolver nuestro problema: ¿Qué tipo de retorno se deduce ?

```
std::cout << "El mayor de " << small_int << " y 19.6 es "  
<< find_max(small_int, 19.6) << std::endl;
```

# Plantillas de funciones

## Deducción de tipo de retorno para plantillas

Volvamos al problema que tenemos para encontrar una nueva solución mas robusta:

```
template <typename T1, typename T2>
??? find_max(T1 a, T2 b)
{
    return a > b ? a : b;
}
```

No hay una manera sencilla de especificar que tipo debe ser devuelto aquí. Pero hay una manera fácil de hacer que el compilador lo deduzca después de instanciar la plantilla:

```
template <typename T1, typename T2>
auto find_max(T1 a, T2 b)
{
    return a > b ? a : b;
}
```

# Plantillas de funciones

## Deducción de tipo de retorno para plantillas

Con esta modificación, nuestro problema queda resuelto, ya que el compilador deduce satisfactoriamente el tipo de retorno.

```
std::cout << "El mayor de " << small_int << " y 9.6 es "  
<< find_max(small_int, 9.6) << std::endl; // tipo de retorno deducido:  
                                // double
```

La deducción del tipo de retorno de función se introdujo en C++ 14.

# Plantillas de funciones

## decltype()

Antes, los creadores de plantillas tenían que recurrir a otros medios cuando el tipo de retorno de una plantilla de función es el de una expresión que depende de uno o más argumentos de tipo de plantilla. En nuestro ejemplo, la plantilla `find_max()`, esta expresión es la expresión `a > b ? : b`. Entonces, sin la deducción del tipo de retorno, ¿cómo podría derivar un tipo de una expresión y cómo podría usar esto en una especificación de plantilla de función?

La palabra clave `decltype` proporciona la solución, al menos en parte. `decltype(expression)` produce el tipo del resultado de evaluar `expression`.

```
template <typename T1, typename T2>
auto find_max(T1 a, T2 b) -> decltype(a > b ? a : b)
{
    return a > b ? a : b;
}
```

# Plantillas de funciones

## `decltype()`

Poner la palabra clave `auto` antes del nombre de la función no siempre le dice al compilador que se debe deducir el tipo de retorno. En su lugar, también puede indicar que la especificación del tipo de devolución vendrá al final del encabezado de la función. Se escribe este tipo de retorno final siguiendo la flecha, `->`, después de la lista de parámetros. En el contexto de las plantillas, el especificador `decltype()` no solo es útil en los tipos de devolución finales, donde su uso ha sido reemplazado principalmente por la deducción del tipo de devolución. Suponga que necesita una función de plantilla para generar la suma de los productos de los elementos correspondientes en dos vectores del mismo tamaño. Luego puede definir la plantilla de esta manera (usando la plantilla `std::min()` definida en `<algorithm>`):

# Plantillas de funciones

## decltype()

```
template<typename T1, typename T2>
auto vector_product(const std::vector<T1>& data1, const std::vector<T2>& data2)
{
    // protección contra vectores de diferente tamaño.
    const auto count = std::min(data1.size(), data2.size());

    decltype(data1[0]*data2[0]) sum {};
    for (size_t i {}; i < count; ++i)
        sum += data1[i] * data2[i];

    return sum;
}
```

Sin decltype(), tendría dificultades para especificar un tipo adecuado para la variable de suma, especialmente si también desea admitir vectores vacíos. Tenga en cuenta que decltype() nunca evalúa realmente la expresión a la que se aplica. La expresión es solo hipotética, utilizada por el compilador para obtener un tipo en tiempo de compilación. Por lo tanto, es seguro usar la plantilla anterior también con vectores vacíos.

# Ejercitación

## Ejercicio:

Escriba su propia versión de la familia de funciones `std::size()` llamada `my_size()` que funciona no solo para arreglos de tamaño fijo sino también para objetos `std::vector<>` y `std::array<>`. No está permitido usar el operador `sizeof()`

# Ejercitación

## Ejercicio:

En C++17, el encabezado `algorithm` de la biblioteca estándar tiene incluida la plantilla de función `std::clamp()`. La expresión `clamp(a,b,c)` se usa para clampedar el valor `a` a un intervalo cerrado dado  $[b,c]$ . Es decir, si `a` es menor que `b`, el resultado de la expresión será `b`; y si `a` es mayor que `c`, el resultado será `c`; de lo contrario, si `a` se encuentra dentro del intervalo  $[b,c]$ , `clamp()` simplemente devuelve `a`. Escriba su propia plantilla de función `my_clamp()` y pruébela con un pequeño programa de prueba

# Archivos de programa y directivas de preprocessamiento

Su usted desea incluir los archivos de cabecera creados por usted en su proyecto, debe incluirlos de la siguiente manera:

```
#include "mi_header.h"
```

## Prevención de la duplicación del contenido del archivo de cabecera

Un archivo de encabezado que incluye en un archivo fuente puede contener directivas `#include` propias, y este proceso puede tener muchos niveles de profundidad. Esta característica se usa mucho en programas grandes y en los encabezados de la biblioteca estándar.

# Archivos de programa y directivas de preprocessamiento

## Prevención de la duplicación del contenido del archivo de cabecera

Con un programa complejo que involucra muchos archivos de encabezado, existe una buena posibilidad de que un archivo de encabezado pueda ser incluido potencialmente más de una vez en un archivo fuente. En algunas situaciones, esto puede ser incluso inevitable. Sin embargo, la regla de una definición prohíbe que la misma definición aparezca más de una vez en la misma unidad de traducción. Por lo tanto, necesitamos una forma de evitar que esto ocurra.

# Archivos de programa y directivas de preprocessamiento

## Prevención de la duplicación del contenido del archivo de cabecera

Tenga en cuenta que ocasionalmente, un encabezado que se incluye en algún encabezado A.h puede incluso, directa o indirectamente, incluir el encabezado A.h nuevamente. Sin un mecanismo para evitar que el mismo encabezado se incluya en sí mismo, esto introduciría una repetición infinita de #includes, lo que haría que el compilador implosionara en sí mismo.

# Archivos de programa y directivas de preprocessamiento

## Prevención de la duplicación del contenido del archivo de cabecera

Solución:

```
// Archivo de cabecera mi_header.h
#ifndef MIHEADER_H
#define MIHEADER_H
// El código entero de mi_header se ubica aquí.
// Este código será ubicado en el código fuente.
// solamente si MIHEADER_H no ha sido definido previamente.
#endif
```

# Ejercitación

## Ejercitación:

Construya una librería que contenga las siguientes funciones estadísticas:

- Valor medio de una muestra
- Mediana de una muestra
- Varianza muestral
- Desviación estándar de una muestra

Varianza de una muestra( $s^2$ )

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1}$$

Desviación estándar  
de un muestra

$$DE_{muestra} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n - 1}}$$

Cree un nombre de espacio llamado sts y englobe todas las funciones definidas.

Construya el archivo de cabecera que le permitirá utilizar dichas funciones.

# Clases

## Definición de un clase

Para especificar el nombre usaremos la convención de nombres CamelCase.

```
class ClassName
{
    // Código que define los miembros de la clase..
};

class ClassName
{
private:
    // Miembros privados
public:
    // Miembros públicos.
};
```

# Clases

## Definición de un clase

Todos los miembros de una clase son privados por defecto. Existen tres especificadores de acceso en C++: *public*, *private* y *protected*.

El modificador de acceso *protected* es similar al de los modificadores de acceso *private*, la diferencia es que los miembros de la clase declarados como *protected* son inaccesibles fuera de la clase, pero cualquier subclase (clase derivada) de esa clase puede acceder a ellos.

# Clases

## Definición de un clase

Por cuestiones de claridad, vamos a comenzar escribiendo clases simples, por ello realizaremos la implementación de los métodos miembro directamente dentro de la propia definición de la clase.

Sin embargo, a medida que las clases se vuelven más largas y complicadas, tener todas las definiciones de métodos miembro dentro de la clase puede hacer que la clase sea más difícil de administrar y trabajar con ella. El uso de una clase ya escrita solo requiere comprender su interfaz pública (métodos públicos), no cómo funciona la clase debajo del capó. Los detalles de implementación de la función miembro simplemente se interponen en el camino.

# Clases

## Definición de un clase

Afortunadamente, C++ proporciona una forma de separar la parte de "declaración" de la clase de la parte de "implementación". Esto se hace definiendo las funciones miembro de la clase fuera de la definición de la clase. Para hacerlo, simplemente defina las funciones miembro de la clase como si fueran funciones normales, pero prefije el nombre de la clase a la función usando el operador de resolución de alcance o ámbito (::) (igual que para un espacio de nombres).

# Clases

## Definición de un clase

Veamos nuestro primer ejemplo:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Método para calcular volumen de una caja
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## Definición de un clase

Para brindar claridad, la siguiente declaración es equivalente a la anterior dado que los miembros son privados por defecto. Si bien esto es válido, por cuestiones de legibilidad siempre proporcione el especificador de acceso `private`.

```
class Box
{
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Método para calcular volumen de una caja
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## Definición de un clase

En general, puede repetir cualquiera de los especificadores de acceso en una definición de clase tantas veces como desee. Esto le permite colocar variables miembro y funciones miembro en grupos separados dentro de la definición de clase, cada uno con su propio especificador de acceso. Puede ser más fácil ver la estructura interna de una definición de clase si agrupa las variables miembro y las funciones miembro por separado, de acuerdo con sus especificadores de acceso. Observe que todos los campos (variables miembro) están inicializados a 1.0 en la declaración de la clase. El siguiente ejemplo crea una instancia de Box llamada `miCaja` e invoca al método público `volumen`.

```
Box miCaja; // Una caja con todas sus dimensiones a 1.0  
  
std::cout << "Volumen de miCaja: " << miCaja.volume() << std::endl; // Volumen  
// es 1.0
```

# Clases

## Constructores

Un constructor de clase es un tipo especial de método que difiere en algunos aspectos significativos de una método ordinario. Se llama a un constructor cada vez que se define una nueva instancia de la clase. Brinda la oportunidad de inicializar el nuevo objeto a medida que se crea y garantizar que los campos contengan valores válidos. Un constructor de clase siempre tiene el mismo nombre que la clase. `Box()`, por ejemplo, es un constructor de la clase `Box`. Un constructor no devuelve un valor y, por lo tanto, no tiene tipo de retorno. Es un error especificar un tipo de retorno para un constructor.

### Constructor por defecto

Si no define un constructor para una clase, el compilador proporcionará un constructor por defecto. Gracias a este constructor por defecto, la clase `Box` se comporta efectivamente como si estuviera definida de la siguiente manera:

# Clases

## Constructores – Constructor por defecto

```
class Box
{
private:
    double length {1};
    double width {1};
    double height {1};

public:
// Así se vería el constructor por defecto
// que es proporcionado por el compilador.
    Box()
    {
        // Cuerpo del método vacío, no hace nada..
    }
    // Método para calcular volumen de la caja.
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## Constructores - Constructor por defecto

El constructor por defecto es un constructor que no recibe argumentos. El provisto por el compilador tiene el cuerpo del método vacío, por ende no hace nada. Los campos no son inicializados por el constructor por defecto provisto por el compilador. Si no se especifica un valor inicial para una variable miembro de un puntero (`int*`, `const Box*...`) o de tipo fundamental (`doble`, `int`, `bool...`), contendrá un valor basura arbitrario. Tenga en cuenta que tan pronto como defina cualquier constructor, incluso uno no predeterminado con parámetros, el constructor por defecto provisto por el compilador ya no se proporcionará. Hay circunstancias en las que necesita un constructor sin parámetros además de un constructor con parámetros que usted defina. En tal caso usted deberá proporcionar el constructor sin parámetros que necesita.

### Definiendo un constructor

Incorporemos un constructor a la clase `Box`:

# Clases

```
#include <iostream>

// Clase para representar una caja
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructor con parámetros
    Box(double lengthValue, double widthValue, double heightValue)
    {
        std::cout << "Constructor de Box llamado." << std::endl;
        length = lengthValue;
        width = widthValue;
        height = heightValue;
    }
    // Método para calcular el volumen de una caja.
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## Constructores - Constructor por defecto

```
int main()
{
    Box firstBox {80.0, 50.0, 40.0}; // Crear una caja.

    double firstBoxVolume {firstBox.volume()}; // Calcular el volumen de la caja
    .
    std::cout << "El volumen de firstBox es: " << firstBoxVolume << std::endl;

    // Box secondBox; // Error de compilación!!
}
```

El objeto `firstBox` ese crea al ejecutar la siguiente sentencia:

```
Box firstBox {80.0, 50.0, 40.0};
```

Dado que se proveen 3 parámetros se invoca al constructor con 3 parámetros provisto. La sentencia de instanciación de `secondBox` genera un error de compilación porque el constructor sin parámetros no fue provisto. Dijimos anteriormente que una vez que define un constructor, el compilador ya no proporcionará un constructor predeterminado, al menos no de forma predeterminada.

# Clases

## Constructores - Constructor por defecto

### Palabra clave default

Tan pronto como agrega un constructor, cualquier constructor, el compilador ya no define implícitamente un constructor predeterminado. Si aún desea que sus objetos sean construibles por defecto, depende de usted asegurarse de que la clase tenga un constructor predeterminado. Su primera opción, por supuesto, es definir uno usted mismo. Para la clase Box, por ejemplo, todo lo que tendría que hacer es agregar la siguiente definición de constructor en algún lugar de la sección pública de la clase.

```
Box() {} // Constructor por defecto
```

Debido a que las variables miembro de Box ya tienen un valor válido, 1.0, durante su inicialización, no queda nada por hacer en el cuerpo del constructor predeterminado.

# Clases

## Constructores - Constructor por defecto

### Palabra clave default

En lugar de definir un constructor predeterminado con un cuerpo de función vacío, también puede usar la palabra clave `default`. Esta palabra clave se puede usar para indicar al compilador que genere un constructor predeterminado, incluso si hay otros constructores definidos por el usuario presentes. Para `Box`, esto se ve de la siguiente manera:

```
Box() = default; // Constructor por defecto.
```

Si bien una definición explícita de cuerpo vacío y una declaración de constructor predeterminada son casi equivalentes, se prefiere el uso de la palabra clave `default` en el código C++ moderno.

# Clases

## Constructores - Constructor por defecto

```
#include <iostream>

// Clase para representar una caja
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // El programador provee el constructor sin parámetros.
    Box() {};
    // Constructor con parámetros
    Box(double lengthValue, double widthValue, double heightValue)
    {
        std::cout << " Constructor de Box llamado." << std::endl;
        length = lengthValue;
        width = widthValue;
        height = heightValue;
    }
    // Método para calcular el volumen de una caja.
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## Constructores - Constructor por defecto

```
#include <iostream>

// Clase para representar una caja
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // El compilador nos provee el constructor por defecto.
    Box() = default;
    // Constructor con parámetros
    Box(double lengthValue, double widthValue, double heightValue)
    {
        std::cout << " Constructor de Box llamado." << std::endl;
        length = lengthValue;
        width = widthValue;
        height = heightValue;
    }
    // Método para calcular el volumen de una caja.
    double volume()
    {
        return length * width * height;
    }
};
```

# Clases

## **Definición de funciones y constructores fuera de la clase**

Dijimos anteriormente que la definición de una función miembro se puede colocar fuera de la definición de clase. Esto también es cierto para los constructores de clases. Todo esto se puede hacer en un solo archivo si lo desea, pero es mucho más común colocar la definición de clase en un archivo de encabezado y las definiciones de las funciones miembro y los constructores en un archivo fuente correspondiente.

# Clases

## Definición de funciones y constructores fuera de la clase

Archivo de cabecera Box.h

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructores
    Box(double lengthValue, double widthValue, double heightValue);
    Box() = default;
    double volume(); // Método para calcular el volumen de una caja.
};

#endif
```

# Clases

## Definición de funciones y constructores fuera de la clase

Archivo fuente Box.cpp

```
// Box.cpp
#include "Box.h"
#include <iostream>

// Definición de constructor
Box::Box(double lengthValue, double widthValue, double heightValue)
{
    std::cout << "Constructor de Box llamado." << std::endl;
    length = lengthValue;
    width = widthValue;
    height = heightValue;
}

// Método para calcular el volumen de un caja.
double Box::volume()
{
    return length * width * height;
}
```

El nombre de cada función miembro y constructor en el código fuente debe estar calificado con el nombre de la clase para que el compilador sepa a qué clase pertenezcan.

# Clases

## Definición de funciones y constructores fuera de la clase

Si Box.h no se incluyera en Box.cpp, el compilador no sabría que Box es una clase, por lo que el código no se compilaría. Tenga en cuenta que un constructor predeterminado que usa la palabra clave **default** en la definición de clase no debe tener una definición en el archivo fuente. Separar las definiciones de clases de las definiciones de sus miembros hace que el código sea más fácil de administrar. Para una clase grande con muchos métodos miembro y constructores, sería muy engorroso si todas las definiciones de funciones aparecieran dentro de la clase. Más importante aún, cualquier archivo fuente que cree objetos de tipo Box solo necesita incluir el archivo de encabezado Box.h.

# Clases

## Definición de funciones y constructores fuera de la clase

Un programador que usa esta clase no necesita acceso a las definiciones de código fuente de las funciones miembro, solo a la definición de clase en el archivo de encabezado. Mientras la definición de la clase permanezca fija, puede cambiar las implementaciones de las funciones miembro sin afectar la operación de los programas que usan la clase.

*Definir una función miembro fuera de una clase en realidad no es lo mismo que colocar la definición dentro de la clase. Una diferencia sutil es que las definiciones de función dentro de una definición de clase son implícitamente inline (aunque esto no significa necesariamente que se implementarán como funciones inline; el compilador decide cual).*

# Clases

## Valores predeterminados de los parámetros del constructor

Cuando discutimos las funciones "ordinarias", vio que puede especificar valores predeterminados para los parámetros en el prototipo de función. Puede hacer esto para funciones de miembros de clase, incluidos los constructores. Los valores de parámetros predeterminados para constructores y funciones miembro siempre van dentro de la clase, no en una definición de función o constructor externo. Podemos cambiar la definición de clase en el ejemplo anterior a lo siguiente:

```
class Box
{
private:
    double length, width, height;
public:
    // Constructores
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
    Box() = default;

    double volume(); // Método que calcula el volumen de una caja.
};
```

# Clases

## Valores predeterminados de los parámetros del constructor

Si hace este cambio al ejemplo anterior, ¿qué sucede? ¡Obtiene un mensaje de error del compilador, por supuesto! El mensaje básicamente dice que tiene varios constructores predeterminados definidos. El motivo de la confusión es que el constructor con tres parámetros permite omitir los tres argumentos, lo que es indistinguible de una llamada al constructor sin argumentos. Un constructor para el que todos los parámetros tienen un valor predeterminado aún cuenta como un constructor predeterminado. La solución obvia es deshacerse del constructor predeterminado que no acepta parámetros en esta instancia. Si lo hace, todo se compila y se ejecuta correctamente.

# Clases

## Uso de una lista de inicializadores de miembros

Hasta ahora, ha establecido valores para las variables miembro en el cuerpo de un constructor utilizando una asignación explícita. Puede usar una técnica alternativa y más eficiente que usa una lista de inicializadores de miembros. Ilustraremos esto con una versión alternativa del constructor de la clase Box:

```
// Definición del constructor utilizando lista de inicializadores de miembros.  
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}  
{  
    std::cout << "Constructor de Box llamado." << std::endl;  
}
```

# Clases

## Uso de una lista de inicializadores de miembros

Los valores de las variables miembro se especifican como valores de inicialización en la lista de inicialización que forma parte del encabezado del constructor. `length` se inicializa con `lv`, por ejemplo. La lista de inicialización está separada de la lista de parámetros por dos puntos (:), y cada inicializador está separado del siguiente por una coma (,). Si sustituye esta versión del constructor en el ejemplo anterior, verá que funciona igual de bien. Sin embargo, esto es más que una notación diferente. Cuando inicializa una variable miembro usando una declaración de asignación en el cuerpo del constructor, la variable miembro se crea primero (usando una llamada al constructor si es una instancia de una clase) después de lo cual la asignación se lleva a cabo como una operación separada.

# Clases

## Uso de una lista de inicializadores de miembros

Cuando usa una lista de inicialización, el valor inicial se usa para inicializar la variable miembro a medida que se crea. Este puede ser un proceso mucho más eficiente, particularmente si la variable miembro es una instancia de clase. Esta técnica para inicializar parámetros en un constructor es importante por otra razón. Como verá, es la única forma de establecer valores para ciertos tipos de variables miembro (constantes, referencias). Hay una pequeña advertencia a tener en cuenta. *El orden en el que se inicializan las variables miembro está determinado por el orden en el que se declaran en la definición de la clase, no por el orden en que aparecen en la lista de inicializadores de miembros, como cabría esperar.*

# Clases

## **Uso de una lista de inicializadores de miembros**

Esto solo importa, por supuesto, si las variables miembro se inicializan usando expresiones para las que importa el orden de evaluación. Los ejemplos plausibles serían donde una variable miembro se inicializa usando el valor de otra o llamando a una función miembro que se basa en otras variables miembro que ya se han inicializado. Confiar en este orden de evaluación en el código de producción puede ser peligroso. ¡Incluso si todo funciona correctamente hoy, el próximo año alguien puede cambiar el orden de declaración y sin darse cuenta romper la corrección de uno de los constructores de la clase!

# Clases

## Uso de una lista de inicializadores de miembros

*Como regla, prefiera inicializar todas las variables miembro en la lista de inicializadores de miembros del constructor. esto es generalmente más eficiente. para evitar cualquier confusión, lo ideal es colocar las variables miembro en la lista de inicializadores en el mismo orden en que se declaran en la definición de clase. Debe inicializar las variables miembro en el cuerpo del constructor solo si se requiere una lógica más compleja o si es importante el orden en que se inicializan.*

# Clases

## **Uso de la palabra clave explicit**

Un problema con los constructores de clase con un solo parámetro es que el compilador puede usar dicho constructor como una conversión implícita del tipo del parámetro al tipo de clase. Esto puede producir resultados no deseados en algunas circunstancias. Consideremos una situación particular. Suponga que define una clase que define cajas que son cubos para los cuales los tres lados tienen la misma longitud:

# Clases

## Uso de la palabra clave explicit

```
#ifndef CUBE_H_
#define CUBE_H_

// Cube.h
class Cube
{
private:
    double side;

public:
    Cube(double aSide); // Constructor
    double volume(); // Calcula volumen de un cubo
    bool hasLargerVolumeThan(Cube aCube); //Compara el volumen de un cubo con otro
};

#endif /* CUBE_H_ */
```

# Clases

## Uso de la palabra clave explicit

```
//Cube.cpp
Cube::Cube(double aSide) : side{aSide}
{
    std::cout << "Constructor de cubo invocado." << std::endl;
}

double Cube::volume() {
    return side * side * side;
}

bool Cube::hasLargerVolumeThan(Cube aCube) {
    return volume() > aCube.volume();
}
```

El constructor requiere solo un argumento de tipo doble. Claramente, el compilador podría usar el constructor para convertir un valor double en un objeto Cube, pero ¿bajo qué circunstancias es probable que suceda? La clase define una función de volumen () y una función para comparar el objeto actual con otro objeto de cubo pasado como argumento, que devuelve verdadero si el objeto actual tiene el mayor volumen. Puede usar la clase Cube de la siguiente manera:

# Clases

## Uso de la palabra clave explicit

```
#include <iostream>
#include "Cube.h"

int main()
{
    Cube box1 {7.0};
    Cube box2 {3.0};

    if (box1.hasLargerVolumeThan(box2))
        std::cout << "box1 es mayor que box2." << std::endl;
    else
        std::cout << "Volumen de box1 es menor o igual que box2." << std::endl;

    std::cout << "volumen de box1 es " << box1.volume() << std::endl;

    if (box1.hasLargerVolumeThan(50.0))
        std::cout << " Volumen de box1 es mayor a 50" << std::endl;
    else
        std::cout << "Volumen de box1 es menor o igual a 50" << std::endl;
}
```

# Clases

## Uso de la palabra clave explicit

La salida que obtenemos del código anterior es:

```
Constructor de cubo invocado.  
Constructor de cubo invocado.  
volumen de box1 is 343  
Constructor de cubo invocado.  
Volumen de box1 es menor o igual a 50
```

La salida muestra que el volumen de box1 definitivamente no es inferior a 50, pero la última línea de salida indica lo contrario. El código supone que hasLargerVolumeThan() compara el volumen del objeto actual con 50.0. En realidad, la función compara dos objetos Cube. El compilador sabe que el argumento de la función hasLargerVolumeThan() debería ser un objeto Cube, pero lo compila muy bien porque hay un constructor disponible que convierte el argumento 50.0 en un objeto Cube. El código del compilador produce es equivalente a lo siguiente:

# Clases

## Uso de la palabra clave explicit

```
if (box1.hasLargerVolumeThan(Cube{50.0}))  
    std::cout << " Volumen de box1 es mayor a 50" << std::endl;  
else  
    std::cout << "Volumen de box1 es menor o igual a 50" << std::endl;
```

La función no compara el volumen del objeto box1 con 50.0, sino con 125000.0, que es el volumen de un objeto Cube con un lado de longitud 50.0. El resultado es muy diferente de lo esperado. Puede evitar que ocurra esto declarando el constructor como explícito:

```
class Cube  
{  
private:  
double side;  
  
public:  
    Cube(double aSide); // Constructor  
  
    double volume(); // Calcula volumen de un cubo  
    bool hasLargerVolumeThan(Cube aCube); //Compara el volumen de un cubo con otro  
};
```

# Clases

## Uso de la palabra clave explicit

El compilador nunca usa un constructor declarado como explícito para una conversión implícita; solo se puede usar explícitamente en el programa. Al usar la palabra clave explícita con constructores que tienen un solo parámetro, evita conversiones implícitas del tipo de parámetro al tipo de clase. El miembro `hasLargerVolumeThan()` solo acepta un objeto `Cube` como argumento, por lo que llamarlo con un argumento de tipo `double` no compila.

*Las conversiones implícitas pueden dar lugar a un código confuso; la mayoría de las veces se vuelve mucho más obvio por qué se compila el código y qué hace si usa conversiones explícitas. De forma predeterminada, debe declarar todos los constructores de un solo argumento como explícitos (tenga en cuenta que esto incluye constructores con múltiples parámetros donde al menos todos menos el primero tienen valores predeterminados); omita explicit solo si las conversiones de tipos implícitas son realmente deseables.*

# Clases

## Delegación de constructores

Una clase puede tener varios constructores que proporcionen diferentes formas de crear un objeto. El código de un constructor puede llamar a otro de la misma clase en la lista de inicialización. Esto puede evitar repetir el mismo código en varios constructores. Aquí hay una ilustración simple de esto usando la clase Box:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructores
    Box(double lv, double wv, double hv);
    explicit Box(double side); // Constructor para un cube
    Box() = default;          // Constructor sin argumentos
    double volume();           // Método para calcular el volumen de una caja.
};
```

# Clases

## Delegación de constructores

Observe que hemos restaurado los valores iniciales de los campos y eliminado los valores predeterminados de los parámetros del constructor. Esto se debe a que el compilador no podría distinguir entre una llamada del constructor con un solo parámetro y una llamada del constructor con tres parámetros con los dos últimos argumentos omitidos. Esto elimina la capacidad de crear un objeto sin argumentos, y el compilador no proporcionará el valor predeterminado, por lo que hemos agregado la definición del constructor sin argumentos a la clase. La implementación del primer constructor puede ser la siguiente:

```
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}  
{  
    std::cout << "Constructor 1 de Box llamado." << std::endl;  
}
```

# Clases

## Delegación de constructores

El segundo constructor crea un objeto Box con todos los lados iguales y podemos implementarlo así:

```
Box::Box(double side) : Box{side, side, side}  
{  
    std::cout << "Constructor 2 de Box llamado." << std::endl;  
}
```

Este constructor simplemente llama al constructor anterior en la lista de inicialización. El argumento `side` se usa como los tres valores en la lista de argumentos del constructor anterior. Esto se denomina **constructor delegado** porque delega el trabajo de construcción al otro constructor. Los constructores delegados ayudan a acortar y simplificar el código del constructor y pueden hacer que la definición de la clase sea más fácil de entender.

# Clases

## Delegación de constructores

Solo debería llamar a un constructor para la misma clase en la lista de inicialización de un constructor. No es lo mismo llamar a un constructor de la misma clase en el cuerpo de un constructor delegante. Además, no debe inicializar variables miembro en la lista de inicialización de un constructor delegado. El código no se compilará si lo hace. Puede establecer valores para las variables miembro en el cuerpo de un constructor delegado pero, en ese caso, debe considerar si el constructor debe implementarse realmente como un constructor delegado.

**Ejercicio:** Realice un ejemplo utilizando los 2 constructores vistos en esta sección, cree un objeto Box y un objeto Cube.

# Clases

## Ejercicio:

Construya una clase Circulo que posea un constructor por defecto y un parametrizado. Implemente los constructores y los métodos perímetro y superficie. Cree al menos dos instancia de dicha clase y calcule el perímetro y superficie de todos los objetos instanciados. Imprima los resultados.

# Clases

## Constructor de copia

Suponga el siguiente fragmento de código:

```
Box box2 {7.0};  
Box box3 {box2};  
  
std::cout << "box3 volume = " << box3.volume() << std::endl; // Volumen = 125
```

El resultado muestra que `box3` tiene las dimensiones de `box2`, pero no hay un constructor definido con un parámetro de tipo `Box`, entonces, ¿cómo se creó `box3`? La respuesta es que el compilador proporcionó un **constructor de copia predeterminado**, que es un constructor que crea un objeto copiando un objeto existente. El constructor de copia predeterminado copia los valores de las variables miembro del objeto que es el argumento del nuevo objeto.

# Clases

## Constructor de copia

El comportamiento predeterminado está bien en el caso de los objetos Box, pero puede causar problemas cuando una o más variables miembro son punteros. Simplemente copiar un puntero no duplica lo que apunta, lo que significa que cuando el constructor de copia crea un objeto, está interrelacionado con el objeto original. Ambos objetos contendrán un miembro que apunta a lo mismo. Un ejemplo simple es si un objeto contiene un puntero a una cadena. Un objeto duplicado tendrá un miembro que apunta a la misma cadena, por lo que si la cadena se cambia para un objeto, se cambiará para el otro. Esto no suele ser lo que quieras. En este caso, debe definir un constructor de copia. Mas adelante profundizaremos aún mas en el constructor de copia.

# Clases

## Constructor de copia - Implementación

El constructor de copias debe aceptar un argumento del mismo tipo de clase y crear un duplicado de manera adecuada. Esto plantea un problema inmediato que debe superar; puede verlo claramente si intenta definir el constructor de copia para la clase Box de esta manera:

```
Box::Box(Box box) : length {box.length}, width {box.width}, height {box.height} // Mal!!  
{}
```

Cada variable miembro del nuevo objeto se inicializa con el valor del objeto que es el argumento. No se necesita código en el cuerpo del constructor de copia en esta instancia.

# Clases

## Constructor de copia - Implementación

Esto se ve bien, pero considere lo que sucede cuando se llama al constructor. El argumento se pasa por valor, pero debido a que el argumento es un objeto Box, el compilador se las arregla para llamar al constructor de copias de la clase Box para hacer una copia del argumento. Por supuesto, el argumento de esta llamada del constructor de copias se pasa por valor, por lo que se requiere otra llamada al constructor de copias, y así sucesivamente. En resumen, ha creado una situación en la que se producirá un número ilimitado de llamadas recursivas al constructor de copia. Su compilador no permitirá que este código se compile. Para evitar el problema, el parámetro del constructor de copia debe ser una referencia. Más específicamente, debería ser un parámetro de referencia a constante. Para la clase Box, esto se ve así:

```
Box::Box(const Box& box) : length {box.length}, width {box.width}, height {box.height}  
{}
```

# Clases

## Constructor de copia - Implementación

Ahora que el argumento ya no se pasa por valor, se evitan las llamadas recursivas del constructor de copias. El compilador inicializa el parámetro `box` con el objeto que se le pasa. El parámetro debe ser referencia a `const` porque un constructor de copias solo se dedica a crear duplicados; no debe modificar el original. Un parámetro de referencia a `const` permite copiar objetos `const` y `non-const`. Si el parámetro fuera una referencia a no constante, el constructor no aceptaría un objeto constante como argumento, por lo que no permitiría la copia de objetos constantes. Puede concluir de esto que el tipo de parámetro para un constructor de copias es siempre una referencia a un objeto `const` del mismo tipo de clase. En otras palabras, la forma del constructor de copias es la misma para cualquier clase:

# Clases

## Constructor de copia - Implementación

```
Type::Type(const Type& object)
{
    // Código para duplicar el objeto
}
```

Por supuesto, el constructor de copias también puede tener una lista de inicialización e incluso puede delegar a otros constructores que no sean copias. Aquí hay un ejemplo:

```
Box::Box(const Box& box) : Box{box.length, box.width, box.height}
{}
```

# Clases

## Constructor de copia - Implementación

**Ejercicio:** Implemente el constructor de copia para su clase Circulo, realice un prueba funcional creando 2 objetos adicionales por copia.

# Clases

## Acceso a miembros de clase privados

Puede proporcionar acceso a los valores de las variables miembro privadas agregando funciones miembro para devolver sus valores. Para proporcionar acceso a las dimensiones de un objeto Box desde fuera de la clase, solo necesita agregar tres funciones a la definición de la clase. Estos método suelen llamarse getters.

Habrá situaciones en las que desee permitir que las variables miembro se cambien desde fuera de la clase. Si proporciona una función miembro para hacer esto en lugar de exponer la variable miembro directamente, tiene la oportunidad de realizar comprobaciones de integridad en el valor. Estos método suelen llamarse setters. Veamos un ejemplo completo:

# Clases

## Acceso a miembros de clase privados

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructores
    Box() = default;
    Box(double length, double width, double height);

    double volume(); // // Método para calcular el volumen de una caja.

    // Métodos para proveer acceso a los valores de miembros privados. getters
    double getLength() { return length; }
    double getWidth() { return width; }
    double getHeight() { return height; }

    // Métodos para modificar los valores de los campos privados. setters
    void setLength(double lv) { if (lv > 0) length = lv; }
    void setWidth(double wv) { if (wv > 0) width = wv; }
    void setHeight(double hv) { if (hv > 0) height = hv; }

};
```

# Clases

## Acceso a miembros de clase privados

Ejemplo de utilización de la clase con métodos getters y setters.

```
Box myBox {3.0, 4.0, 5.0};  
std::cout << "Las dimensiones de myBox son " << myBox.getLength()  
<< " x " << myBox.getWidth()  
<< " x " << myBox.getHeight() << std::endl;  
  
myBox.setLength(-20.0); // valor ignorado.  
myBox.setWidth(40.0);  
myBox.setHeight(10.0);  
  
std::cout << "Las dimensiones de myBox ahora son " <<  
myBox.getLength()  
<< " x " << myBox.getWidth()  
<< " x " << myBox.getHeight() << std::endl;
```