C++ SEMANA 2

Smart pointers (punteros inteligentes)

Como hemos visto, la asignación dinámica de memoria conlleva varios riegos: 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 comportamientos 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)

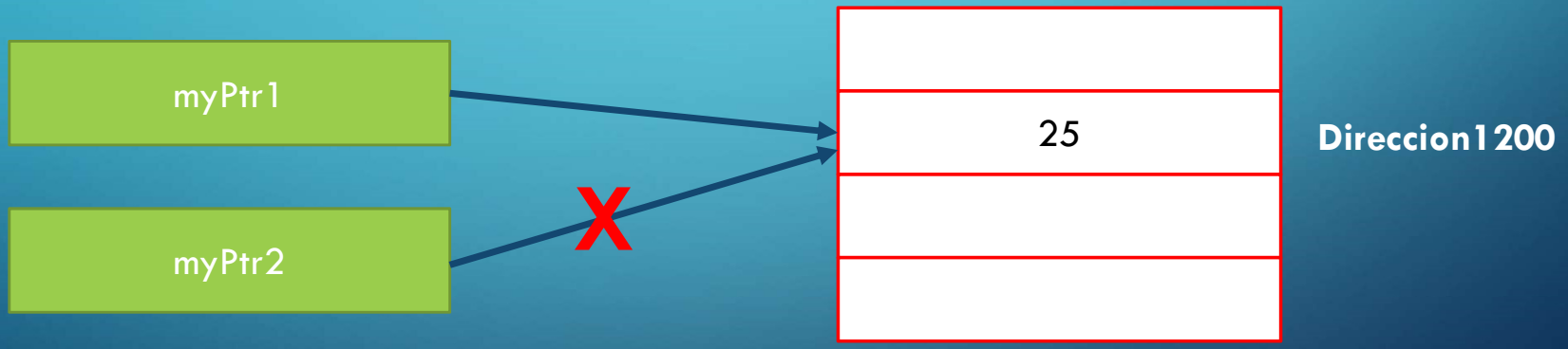
- Un objeto unique_ptr<T> se comporta como un puntero al tipo T y es "único" en el sentido de que solo puede haber un único objeto unique_ptr<> que contenga la misma dirección. En otras palabras, nunca puede haber dos o más objetos unique_ptr<T> apuntando a la misma dirección de memoria al mismo tiempo. 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)

Usando punteros unique_ptr<T>

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 esta 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.

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

std::unique_ptr<int>myPtr2 = std::move(myPtr1);
std::cout << *myPtr2 << std::endl;
std::cout << *myPtr1 << std::endl;   // Error. myPtr1
                                     // apunta a nada
```

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
                                               // dinamicamente
```

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 mas versátil y poderoso. Se presentan ejemplos simples con arrays para simplificar la explicación de este tipo de punteros. Se 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 direccion 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<>** 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>

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(); // Not es 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.

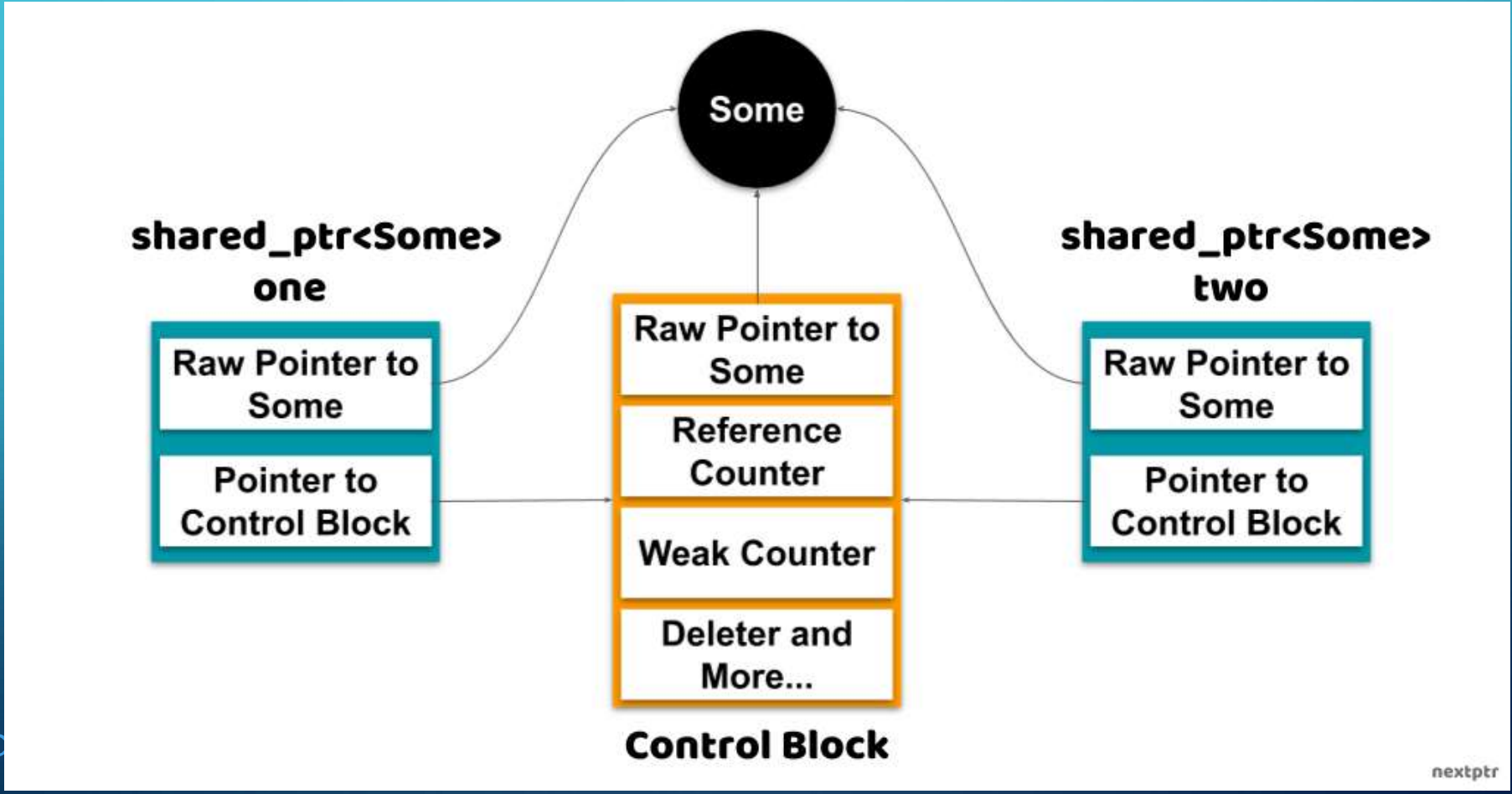
Smart pointers (punteros inteligentes)

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:

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>

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;
}
```

Definición

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

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 traves
 de []. El parametro de funcion 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 asi:

```
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, calculado 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 asi:

```
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, calculado 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 implicitas

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 implicitas

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 implicitas

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 implicitas

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 implicitas

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

El 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

Retornando valores de una función – Retornando un puntero

```
// Retorna el minimo de la muestra
const double *find_min(const double *samples, size_t size)
{
    const double *ptr_success{};// para establecer un solo punto de retorno.

    if(size){

        size_t ind_min{};

        for(size_t ind{}; ind < size; ++ind)
        {
            if(samples[ind_min] > samples[ind])
                ind_min = ind;
        }
        ptr_success = &samples[ind_min];
    }
    return ptr_success;   // es mas sencillo devolver el indice,
                          // esto se hace asi para ejemplificar
                          // la devolucion de punteros.
}
```

Funciones

Retornando valores de una función – Retornando un puntero

```
// Retorn el maximo de la muestra
const double *find_max(const double *samples, size_t size)
{
    const double *ptr_success{};// para establecer un solo punto de retorno.

    if(size){

        size_t ind_max{};

        for(size_t ind{}; ind < size; ++ind)
        {
            if(samples[ind_max] < samples[ind])
                ind_max = ind;
        }

        ptr_success = &samples[ind_max];
    }
    return ptr_success;   // es mas sencillo devolver el indice,
                          // esto se hace asi para ejemplificar
                          // la devolucion de punteros.
}
```

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.