Introducción a C++ moderno

Por Jose Antonio Verde Jiménez y
Luis Daniel Casais Mezquida

Grupo de Usuarios de Linux
@guluc3m | gul.uc3m.es

Transparencias

github.com/guluc3m/modern-cpp

C++ Moderno

- Más que «C con cosas»
  - Más rápido
  - Más seguro (memoria, tipado, ...)
  - Mejores abstracciones
  - Soporte para programación genérica y funcional
- Usar las cosas nuevas es *opt-in*

PROHIBIDO

- *Raw pointers* - `int* ptr;`
- *C arrays* - `int[] arr;`
- Manipulación manual de memoria - `new` / `malloc()`
- Librerías de C - `#include <whatever.h>`

Todo esto hace que el código no sea seguro.

Estándares

Especificaciones ISO de una versión del lenguaje.

- Documentación en cppreference.com
- Los compiladores son los encargados de implementarlos
  - Nadie les "obliga"
  - Puedes ver la compatibilidad en cppreference
- Uno nuevo cada 3 años
  - Actualmente: C++23 (Sept. 2024)

Se considera "C++ moderno" a partir de C++17.

CppCoreGuidelines

- Guía de codificación de C++
  - Más seguro
  - Pilla más errores a la hora de compilar
  - Por el mismísmo e inigualable Bjarne Stroustrup
- `gsl::span` vs. `std::span`

clang-tidy

- Herramienta de clang
  - Comprueba que el código se adhiera a una guía de codificación
  - Permite capturar muchos errores en tiempo de compilación
  - Homogeneiza el código escrito por varias personas
- Añade una capa más de seguridad sobre C++
- Cualquier proyecto que se respete debe tener una guía de codificación

clang-format

- Aplica automáticamente un formato al código
- Todo el código fuente tiene la misma estructura

Si no, cada uno escribe el código como le parezca:

```
    bool is_prime(int x){for(int i=2;i*i<=x;++i){
if(x%i==0)return false;} return true;}
 std::vector<int> prime_numbers ( int   from,   int to )
{ std::vector<int>   result
; for (   int i = from ;    i < to; ++i)
  { if (is_prime(i)) {    result.push_back(i)   }
   }
; return result;}
```

Con clang-format:

```
bool is_prime (int x) {
   for (int i = 2; i * i <= x; ++i) {
      if (x % i == 0) { return true; }
   }
   return false;
}

std::vector<int> prime_numbers (int from, int to) {
   std::vector<int> result;
   for (int i = from; i < to; ++i) {
      if (is_prime(i)) { result.push_back(i); }
   }
   return result;
}
```

Cosas nuevas

`auto`

Permite al compilador inferir el tipo de una variable.

```
auto x = 1; // int
```

Útil al recorrer vectores:

```
for (auto it = vector.begin(); it != vector.end(); ++it) {
    it->doSomething();
}
```

`enum class`

Evolución de `enum`, quitando ciertas limitaciones.

```
enum class Color {
    Red,
    Green,
    Blue
};

auto col = Color::Red;
```

`using`

Evolución de `typedef`.

```
using Address = std::uint32_t;
```

Ranged for loops

Una forma más elegante de recorrer contenedores.

```
for (auto & elem : myVec) { }
```

Puedes desempaquetar valores:

```
for (auto & [key, value] : myMap) { }
```

Recuerda usar `const` si no vas a modificar los elementos.

Excepciones

Existen. Casi mejor que ni las toques.

Tienes `try` - `catch`:

```
try {
   foo[index];
} catch (std::out_of_range &e) {
   throw std::exception("cosa mala")
}
```

Constructores

Se ejecutan al instanciar una clase.

- Permiten inicializar los miembros antes de la clase (`:`)

E.g.:

```
class Foo {
   public:
      Foo(int a, int b) : a_ {a}, b_ {b} { /* body */ }

   private:
      int a_;
      int b_;
}

auto bar = Foo {1, 2};
```

Casting

Consiste en convertir información de un tipo de dato a otro. E.g. `int` → `long`

En C:

```
int foo = 69;
long var = (long) foo;
```

Sin embargo, esto es **completamente inseguro**.

C++ provee alternativas, las cuales *destacan*:

- `reinterpret_cast`
- `static_cast`

`reinterpret_cast`

Permite *reinterpretar* los datos entre punteros de distintos tipos.

En C:

```
float x = 42.0;
float *p_x = &x;

unsigned *p_y = (unsigned*)p_x;
// `p_y' apunta a la misma dirección que `p_x' (que es `x').
// Pero para `p_x' la dirección es de tipo `float',
// y para `p_y' la dirección es de tipo `unsigned int'

unsigned y = *p_y;
printf("0x%08X\n", y); // Imprime 0x42280000
```

En C++:

```
float x = 42.0;
float *p_x = &x;

unsigned *p_y = reinterpret_cast<unsigned*>(p_x);

unsigned y = *p_y;
std::cout << std::hex << x << "\n"; // Imprime 42
```

Hay que tener mucho cuidado:

- Hay requisitos de tamaño y de alineamiento
- No se puede utilizar con todos los tipos
- CppCoreGuidelines **lo prohíbe**
    - Se debe justificar su uso

`static_cast`

Hace una conversión «real» entre tipos.

- `static_cast<int>(42.3)` devuelve `42`
- `static_cast<float>(32)` devuelve `32.0`

Si existe un operador de conversión, se utiliza:

- `static_cast<bool>(my_file)` devuelve o `true` o `false`

Hay que tener cuidado con ciertas conversiones (clang-tidy ayuda):

```
static_cast<unsigned>(-3);          // No se puede representar
static_cast<float>(33'554'432);     // Pierde información
static_cast<int>(1.2E100);          // Demasiado grande
static_cast<int>(7'000'000'000L); // Demasiado grande
```

Entrada y Salida

Hay dos clases principales:

- `std::istream` (Flujo de entrada)
- `std::ostream` (Flujo de salida)

Por defecto se utilizan para leer y escribir texto.

Bibliotecas dependiendo del uso:

- `<iostream>`: stdin/stdout
- `<fstream>`: Ficheros

```cpp
#include <iostream>  // Incluye flujos de entrada y salida
                     // (I/O Stream)
#include <string>    // Contiene el tipo std::string

int main () {
   // Redirige la cadena "¿Cuántos años tienes?\n" al flujo
   // de salida estándar `std::cout'
   std::cout << "¿Cuántos años tienes?\n";
   std::string age;

   // Lee la edad en la variable `age' de entrada estándar
   std::cin >> age;

   // Se pueden imprimir varios tipos:
   std::cout << "Tienes " << age << " años\n";

   return 0;
}
```

Entrada y Salida binaria

Los objetos en memoria se almacenan como una secuencia de bytes.

Por ejemplo:

```
int x = 42;
```

Dependiendo del computador, puede ser:

- *little-endian*: `2A 00 00 00`
- *big-endian*: `00 00 00 2A`

(Vamos a suponer *little-endian*)

Un número de coma flotante de simple precisión se representa en memoria en base al estándar IEEE 754.

```
float x = 42.0;
```

- En hexadecimal es 42280000_{16}
- En memoria se representa como `00 00 28 42`

De igual manera la cadena `"hola"` se representa como la secuencia bytes `'h'`, `'o'`, `'l'`, `'a'`.

- En memoria, `68 6f 6c 61`

En un archivo sabemos escribir *strings*:

```
std::ofstream my_output{"my-file.txt"};

my_output << 42     << "\n"
          << 42.0   << "\n"
          << "hola" << "\n";

my_output.close();
```

Lo que resulta en `my-file.txt`:

```
42
42.0
hola
```

¿Y si en vez de escribir cadenas, escribimos los bytes *a pelo*?

Salida binaria en C++

Empezamos abriendo el archivo, de forma binaria:

```
#include <fstream>
#include <iostream>
```

```
std::ofstream file{"my-file.bin", std::ios::binary};
// También:
//    std::ofstream file;
//    file.open("my-file.bin", std::ios::binary);

if (not file) {   // Comprobamos que se abrió bien
   std::cerr << "No se pudo abrir el archivo\n";
   return -1;
}
```

Ahora podemos escribir distintos valores:

```
int int_value = 42; // ¡Número mágico!
float float_value = 42.0;
std::string string_value = "hola";

file.write(
   reinterpret_cast<const char*>(&int_value),
   sizeof(int_value)
); // ¡Reinterpret cast!
file.write(
   reinterpret_cast<const char*>(&float_value),
   sizeof(float_value)
); // ¡Reinterpret cast!
file.write(
   string_value.data(),
   string_value.size()
); // Conversión implícita
```

peeeero...

¡Clang-tidy se queja!

La función miembro `std::ostream::write` pide:

- Un puntero a una cadena de carácteres (`const char *`)
- El tamaño de la cadena (`std::size_t`).

(similar a la función `fwrite` en C)

En este caso está justificado el uso del `reinterpret_cast` :

- No se puede hacer de ninguna otra forma
- Hay que silenciar el clang-tidy (con `NOLINTNEXTLINE`), justificándolo

```
// NOLINTNEXTLINE (cppcoreguidelines-pro-type-reinterpret-cast)
file.write(
   reinterpret_cast<const char*>(&int_value),
   sizeof(int_value)
);

// NOLINTNEXTLINE (cppcoreguidelines-pro-type-reinterpret-cast)
file.write(
   reinterpret_cast<const char*>(&float_value),
   sizeof(float_value)
);

file.write(string_value.data(), string_value.size());

file.close(); // ¡Recordad cerrar el archivo!
```

- Solo está justificado silenciarlo en lectura y escritura
- En cualquier otro caso, tiene que estar **muy** justificado

Ejemplo completo en `ejemplos/0-serialización`.

Si ejecutamos el programa anterior obtenemos un archivo `my-file.bin`.

Podemos leerlo con `hexdump`:

```
$ hexdump -C my-file.bin
```

```
00000000  2a 00 00 00 00 00 28 42  68 6f 6c 61              |*.....(Bhola|
0000000c
```

Entrada binaria en C++

Es similar a la salida:

- `std::ifstream` en vez de `std::ofstream`
- `.read()` en vez de `.write()`

Recordad manejar los errores, y cerrar el archivo.

```
std::ifstream file{"my-file.bin", std::ios::binary};
if (not file) { /* ... */ }

float float_value;

if (
   not file.read(
      reinterpret_cast<char*>(&float_value),
      sizeof(float_value)
   )
) { /* ... */ }

file.close();
```

Ejemplo completo en `ejemplos/2-deserialización`.

Standard Template Library

Conjunto de contenedores y algoritmos genéricos para ayudar en la programación.

Templates

A veces tenemos que implementar la misma función para distintos tipos:

```
void print_square (int x) {
   std::cout << x * x << std::endl;
}

void print_square (long x) {
   std::cout << x * x << std::endl;
}
```

```
print_square(2);   // Funciona, es `int'
print_square(2L);  // Funciona, es `long'
print_square(2.0); // NO FUNCIONA, es `double'
```

Podemos usar una «plantilla» asumiendo un tipo genérico:

```
template <typename T>
void print_square (T x) {
   std::cout << x * x << std::endl;
}
```

- La `T` se sustituye por el tipo que le pasemos:
  - `print_square<int>(10)`
  - También se deduce el tipo, e.g. `print_square(10)`
- Si para un tipo no se define alguna función, falla

`print_square<float>` es equivalente a:

```
void print_square (float x) {
   std::cout << x * x << std::endl;
}
```

Generalicemos las funciónes de lectura y escritura:

```
template <typename T>
void write (std::ostream & out, T const & value) {
   // NOLINTNEXTLINE (cppcoreguidelines-pro-type-reinterpret-cast)
   out.write(reinterpret_cast<const char*>(&value),
      sizeof(T)
   );
}

// Especializamos para std::string
void write (std::ostream & out, std::string const & value) {
   out.write(
      value.data(),
      static_cast<std::streamsize>(value.size())
   );
}
```

```
template <typename T>
bool read (std::istream & in, T & value) {
   // NOLINTNEXTLINE (cppcoreguidelines-pro-type-reinterpret-cast)
   return static_cast<bool>(in.read(reinterpret_cast<char *>(&value),
      sizeof(T))
   );
}

bool read (std::istream & in, std::string & str, int length) {
   str = std::string(length, 0);
   return static_cast<bool>(in.read(str.data(), length));
}
```

Y ya lo podemos usar en cualquier momento:

```
constexpr int solution{42};
```

```
std::ofstream output_file{"data.bin", std::ios::binary};
if (not output_file) { /* ... */ }

write(output_file, solution);
write(output_file, static_cast<float>(solution));
write(output_file, std::string{"hola"});

output_file.close();
```

```
std::ifstream input_file{"data.bin", std::ios::binary};
if (not input_file) { /* ... */ }

int int_value{};
float float_value{};
std::string string_value{};

if (
   not read(input_file, int_value) or
   not read(input_file, float_value) or
   not read(input_file, string_value, 4))
{ /* ... */ }

std::cout << int_value    << "\n"
          << float_value  << "\n"
          << string_value << "\n";

input_file.close();
```

Ejemplo completo en `ejemplos/3-templates`.

Contenedores

Estructuras de datos genéricas.

Implementan dos funciones miembro básicas:

- `.size()`: Devuelve el tamaño de la estructura
- `.clear()`: Vacía la estructura

`std::vector`

Constructores:

```
std::vector<float> tus_notas(3, 0.0);  // {0.0, 0.0, 0.0}

std::vector<std::string> ciudades {
    "Madrid",
    "Nueva York",
    "París"
};
```

Se pueden recorrer con *ranged for loops*:

```
for (auto const & ciudad : ciudades) {
    std::cout << ciudad << "\n";
}
```

Funciones Miembro:

- `operator[]` y `.at()`: Acceso al elemento
- `.push_back()` y `.emplace_back()`: Añaden un elemento al final del vector
- `.reserve()`: Pre-reservan espacio para el vector

```
#include <vector>
```

```
ciudades.push_back("Murcia");

tus_notas[1] = 0.1;
```

`std::tuple`

Una colección de datos de múltiples tipos.

Útiles para retornar de funciones, ya que se pueden desacoplar elegantemente.

```
#include <tuple>
```

```
std::tuple<std::string, int, double> foo() {
   return {"hola", 42, 42.0};
}

auto [str_v, int_v, flt_v] = foo();
```

`std::map` / `std::unordered_map`

Contiene pares clave-valor con claves únicas.

- `std::map` implementa un árbol de búsqueda binario
- `std::unordered_map` implementa un *hash map*

```
#include <map>
```

```
std::map<int, std::string> ciudades {
   {1, "Madrid"},
   {2, "Nueva York"},
   {3, "París"}
};
```

Funciones miembro:

- `operator[]`: Acceso al valor asociado a la clave.
- `.at()`: Devuelve el valor asociado a la clave.
  Si no existe, lanza una excepción
- `.contains()`: Comprueba si una clave existe

```
ciudades[4] = "Murcia";
std::string mi_fav = ciudades.at(1);
if (ciudades.contains(3)) { /* ... */ }
```

También se pueden iterar y desacoplar fácilmente:

```
for (auto const & [id, nombre] : ciudades) {
   std::cout << nombre << " " << id "\n";
}
```

Algoritmos

Funciones genéricas para cualquier contenedor.

- Ordenar, `std::sort(inicio, fin, predicado)`
- Mapear, `std::transform(inicio, fin, salida, operación)`
- Reducir, `std::accumulate(inicio, fin, primero, función)`
- Filtrar, `std::copy_if(inicio, fin, salida, predicado)`

Las puedes concatenar:

```
bool mayor_a_menor (int x, int y) { return x > y; }
int cuadrado (int x) { return x * x; }
int producto (int a, int b) { return a * b; }

std::vector<int> valores{1, 3, 4, 5, 2};

std::sort(valores.begin(), valores.end(), mayor_a_menor);
std::transform(
   valores.begin(),
   valores.end(),
   valores.begin(),
   cuadrado
);
int result = std::accumulate(
   valores.begin(),
   valores.end(),
   1,
   producto
);
```

Funciones lambda y predicados

Funciones sin nombre, que se crean en el momento.

Útiles en combinación con algoritmos:

```
// bool mayor_a_menor (int x, int y) { return x > y; }

std::sort(
   valores.begin(),
   valores.end(),
   [](int a, int b){ return a > b; }
);
```

```
[capturas] (parámetros) -> retorno { /* código */ }
```

- El retorno es opcional si el compilador lo puede deducir
- Las capturas son más complicadas...
  - Su propósito es cómo se tratan las variables que se referencian en el código

```
auto filtrar_menores_de (std::vector<int> const & v, int n) {
   std::vector<int> resultado;
   std::copy_if(v.begin, v.end(), std::back_inserter(resultado),
      [=] (int x) {  // [=] captura n, copiándola
         return x < n;
      }
   );
   return resultado;
}
```

Funciones

Paso de parámetros

- `T x`: Copia `x`
- `const T x`: Copia `x`, no se puede modificar
- `T & x`: Referencia a `x`, mutable
- `T const & x`: Referencia a `x`, inmutable
- `T && x`: Paso por movimiento

Depende del tipo, se pasa por valor o referencia constante:

- Si es pequeño (`int`, `char`, pequeños `struct`): **copia**
- Si es grande, o se tienen que copiar muchos datos (`std::vector`, `std::string`): **referencia constante**

Especificadores

Contratos con las funciones, para que sean más óptimas.

- `noexcept`: La función no va a lanzar ninguna excepción. Si lo hace... *cagaste*
- `[[nodiscard]]`: Vas a capturar lo que devuelve la función
- `inline`: *A veces*, si la función es lo suficientemente pequeña, se copia en vez de llamarla
- `constexpr`: Son funciones que, por lo general, se ejecutan en tiempo de compilación. No las uses con bloques `try` - `catch`.

```
constexpr bool is_prime (int n) {
   for (int i = 2; i * i <= n; ++i) {
      if (n % i == 0) { return false; }
   }
   return true;
}

template <int length>
constexpr auto calc_primes () {
   std::array<int, length> result;
   int num = 2;
   for (int i = 0; i < length; num++) {
      if (is_prime(num)) { result[i++] = num; }
      else { num; }
   }
   return result;
}
```

Estructuración del código

namespace

Permiten agrupar funciones y clases por funcionalidad.

```
namespace mates {
   constexpr float π = 3.1415'9265'35;
   int cuadrado (int x);
   int raiz (int x);
}
```

- Se puede llamar desde fuera como `mates::cuadrado`.
- Se puede incluir todos los elementos del espacio de nombres con `using`:

```
using namespace mates;
int pitagoras (int x, int y) {
   return raiz(cuadrado(x) + cuadrado(y));
}
```

Header files (`.hpp`) y *source files* (`.cpp`)

En el *header* van las *declaraciones*. En el *source* van las *definiciones* (implementación).

- En el *header* puede haber definiciones pero, por defecto, son *inline*
- Las definiciones de una clase si están dentro del header son **siempre** inline
- Constructores y macros en *header*
- Usa *guards* en los *header* (`#ifndef LIB_HPP`, `#define LIB_HPP`) para prevenir duplicados
- Al definir funciones miembro en el *source*, hay que especificar la clase: `MyClass::my_method() {}`

```cpp
// lib.hpp

#ifndef LIB_HPP
#define LIB_HPP

inline say_hello() { std::cout << "hello\n"; }
int foo(int bar);

class MyClass() {
    public:
        MyClass(float y): _y {y} { }

        int get();

    private:
        int _x = 0;
        float _y;
};

#endif
```

```
// lib.cpp

int foo(int bar) {
    return bar + myVar;
}

int myVar = 0;

MyClass::get() { return _x + _y; }
```

¡Ánimo!

Grupo de Usuarios de Linux

@guluc3m | gul.uc3m.es