C++ moderno

Daniel Pozo Escalona

4/11/17

Estándares de C++

C++ está regulado por varios estándares, entre ellos:

- C++ 03
- C++ 11
- C++ 14
- C++ 17
- C++ 20 (aún no completado)

C++ 03

Es una revisión al estándar C++ 98, y el que más se enseña en la universidad. Tiene muchas carencias que no lo hacen comparable a otros lenguajes modernos, como una biblioteca estándar pequeña, de modo que Boost y otras se usan como tal.

Siguientes estándares

Los primeros cambios que modernizan C++ se dan en este estándar, y cubriremos muchos de ellos. C++ 14 es una revisión a este estándar.

C++ 17 es el último estándar ISO aprobado. El próximo será previsiblemente C++ 20.

Características añadidas en C++ 11

Características añadidas en C++ 11

Ergonomía

Inferencia de tipo

C++ 11 permite el uso de la palabra clave `auto`

```
for(auto i = contenedor_complejo.begin();
    i != contenedor_complejo.end(); i++)
{
  // Código superimportante
}
```

Escribir explícitamente el tipo de `i` puede ser difícil y tedioso.

Inferencia de tipo: `decltype`

EL tipo de una expresión puede ser inferido además en otro contexto, y con resultado potencialmente distinto: al usar `decltype`.

```
int a = 42;
decltype(a) b = a; // int
decltype((a)) c = a; // int&
```

Comparación: inferencia de tipo en Crystal

```
if rand(2) > 0
  foo = "hello world"
end

# foo tiene tipo (String | Nil),
# por lo que esta línea no compila
puts foo.upcase
```

`for` basado en rango

Puede usarse con cualquier objeto que implemente los métodos `begin` y `end` con la semántica usual.

```
for(auto i : contenedor)
{
  // Código
}
```

Soporte de Unicode

C++ 11 define tres tipos de literales de *string* nuevos:

```
u8"Caracteres en UTF-8: \u03BB."
u"Caracteres en UTF-16."
U"Caracteres en UTF-32."
```

Además, cada uno de ellos se puede combinar con el especificador para cadenas *raw*:

```
auto s = u8R"(Cadena de caracteres con "comillas")";
```

Literales definidos por el usuario

Permite definir sufijos que indiquen que ciertos literales se deben construir de una forma especificada por el usuario

```
nn::MLP operator""_mlp(const char* literal, size_t l)
{
  auto capas = extraer_numeros(literal);
  return nn::MLP(capas);
}

auto p = "0.1,5.7,2"_point;
auto nn = "64, 130, 10"_mlp;
```

Ejemplo de uso: biblioteca de JSON

El uso de cadenas *raw* y literales definidos por el usuario puede servir para integrar formatos de texto plano en el lenguaje.

```
auto j = R"(
  {
    "happy": true,
    "pi": 3.141
  }
)"_json;
```

Características añadidas en C++ 11

Semántica de movimiento

Referencias a *rvalues*

En C++ 11 se introducen referencias modificables a *rvalues*, con tipo `T&&`. Estas referencias se usan para evitar copias profundas de objetos temporales.

En C++ 03, el siguiente código hace una de las siguientes cosas:

- Llama al constructor de copia del tipo de `v` con el valor de retorno de la función como argumento.
- Aplica la *optimización de valor de retorno*, que consiste en cambiar el código para que la función tome una referencia a `v` y lo modifique directamente, pudiendo ignorar efectos colaterales del constructor de copia.

```
v = devuelve_vector_gigante();
```

Referencias a *rvalues*

En C++ 11, el tipo de `v` puede definir un constructor que tome una referencia a un objeto temporal, y apropiarse de su estado, normalmente copiando punteros en lugar de vectores completos.

El tipo `std::vector` y otros muchos de la biblioteca estándar implementan un constructor de este tipo.

Características añadidas en C++ 11

Programación orientada a objetos

Inicialización de datos miembros no estáticos

Hasta ahora, había que inicializar todos los datos miembro de una clase explícitamente en el constructor.

```
class Foo {
  int bar = 42;
  int foobar;

public:
  // bar vale 42
  Foo(int foobar) : foobar(foobar) {}
};
```

default y delete

Se añaden palabras clave para indicar explícitamente que una clase va a usar una de las funciones especiales que el compilador provee por defecto (constructor, destructor, constructor de copia, ...) o que no permite llamarlos.

```
class Foo {
public:
  Foo() = default;

  Foo(Foo&) = delete;
  Foo& operator=(Foo&) = delete;

  Foo(int a);
};
```

override y final

También se puede indicar explícitamente que se está redefiniendo un método de una clase base, o que este no se puede redefinir, para que el compilador lo fuerce.

```
class Foo {
public:
  void f() final;
  void f2(std::vector<uint32_t>);
};

class Bar {
public:
  void f(); // Error
  void f2(std::vector<int>) override; // Error
};
```

Características añadidas en C++ 11

Programación funcional

Funciones anónimas

También conocidas como funciones *lambda*.

```
auto id = [](int x) { return x; };
auto sum = [](int x, int y){ return x+y; };
auto curried_sum = [](int x){
  return [=](int y) { return x+y; };
};
```

Clausuras

Permiten que una función anónima use variables definidas en su entorno, por valor o por referencia:

```
int a=0, b=1;

// Toma a por valor y b por referencia
auto f = [a,&b]() { b = 2; };

// Toma todo el entorno por referencia
auto g = [&](){ b = 3; };

// Toma todo el entorno por valor
auto h = [=]() { std::cout << a << std::endl; };
```

Las *lambdas* no son puras

Aunque no se especifique ninguna clausura, una *lambda* puede usar variables globales. Por tanto, no verifican la propiedad de transparencia referencial.

Características añadidas en C++ 11

Seguridad

Constante `nullptr`

Es de tipo `nullptr_t`, que se puede convertir a cualquier tipo de puntero y a `bool`, y a ningún otro tipo integral.

```
void foo(char*);
void foo(int);

// Llama a foo(int)
foo(NULL);
// Llama a foo(char*)
foo(nullptr);
```

Punteros automáticos

Sirven para expresar semántica de pertenencia de un objeto a una o varias referencias, de forma que siempre se determine en qué momento se destruye el objeto y se liberan los recursos asociados.

Esto soluciona el problema de que en C++ los objetos dinámicos no se destruyen al terminar el ámbito en el que fueron creados, lo cual hacía inútil la semántica RAII en estos casos.

Punteros automáticos

`unique_ptr`

Sirve para expresar la pertenencia de un objeto a una única referencia.

```
std::unique_ptr p(new Foo);
// No compila: semántica de
// pertenencia en tiempo de compilación
std::unique_ptr q = p;
```

El destructor de `unique_ptr` destruye el objeto, es decir, implementa la semántica RAII.

```
{
  std::unique_ptr<Foo> p(new Foo);
} // Foo es destruido
```

Punteros automáticos

`shared_ptr`

Sirve para expresar la pertenencia de un objeto a varias referencias. Es un puntero con conteo de referencias, de forma que el puntero que, al ser destruido, ponga la cuenta a cero, es el responsable de destruir el objeto.

Punteros automáticos

```
{
// Cuenta a uno
auto p1 = std::make_shared(new Foo(43));

  {
    // Cuenta a dos
    auto p2(p1);
  } // Cuenta a uno

} // Cuenta a cero: destruye Foo
```

Punteros automáticos

`weak_ptr`

Es una referencia "observadora" que depende de un `shared_ptr`. No afecta al conteo de referencias del `shared_ptr` asociado. Puede observar el estado del objeto (si ha sido destruido) y crear nuevas referencias compartidas a él.

Comparación: semántica de pertenencia en Rust

En Rust, para un objeto:

- Siempre hay una variable que lo posee, y es única.
- Cuando la variable sale de ámbito, el objeto se destruye.

Además:

- Las siguientes posibilidades son exclusivas:
  - Existe una referencia que permite modificar del objeto.
  - Existen varias referencias que no permiten modificar el objeto.
- Cualquier referencia siempre es válida.

Características añadidas en C++ 11

Concurrencia y paralelismo

Concurrencia y paralelismo

En C++ 11 se introducen primitivas de concurrencia y sincronización en la biblioteca estándar.

`thread_local` y `std::thread`

`thread_local` es un nuevo especificador de almacenamiento, que indica que una variable tiene una copia por cada hilo de ejecución del programa. Sin embargo, *no* es una variable local, es global a cada hilo.

`std::thread`: un objeto de este tipo representa un hilo de ejecución.

Ejemplo

`thread_local.cpp`

`future`

Un *futuro* representa un valor posiblemente indeterminado, normalmente porque no ha sido computado, pero que será determinado en algún momento.

La función `std::async` nos permite crear un futuro a partir de un objeto con operador de llamada a función, llamándolo de forma concurrente o diferida.

future

```
for(auto& f : fut)
{
  f = std::async(std::launch::async, generar_solucion,
                 aleatorio());
}

optima = std::accumulate(fut.begin(), fut.end(), def,
            [](sol a, std::future<sol>& b){
            auto v_b = b.get();
            return v_heuristico(a) > v_heuristico(v_b)
            ? a : v_b;});
```

Características añadidas en C++ 11

Otro

Sintaxis estándar para atributos

```
#pragma once // Directiva
 // Sintaxis específica de Microsoft
__declspec(dllimport)
// Sintaxis específica de GNU
__attribute__((constructor))

[[attr_name]] // Sintaxis estandarizada
```

En C++ 14 y 17 se introducen algunos atributos útiles.

Características añadidas en C++ 11

Metaprogramación

Plantillas con número variable de argumentos

```
template<class F1, class F2>
auto compose_mon(F1 f1, F2 f2)
{
  return compose_mon2(f1, f2);
}

template<class F1, class F2, typename... Args>
auto compose_mon(F1 f1, F2 f2, Args... args)
{
  return compose_mon(compose_mon2(f1, f2), args...);
}
```

Aserciones estáticas

Son aserciones que se comprueban en tiempo de compilación, después de la etapa de preprocesamiento.

```
template<typename T> class MiClasse {
  static_assert(std::is_enum<T>::value,
  "T debe ser de al menos 64 bits");
};
```

Características añadidas en C++ 14

`auto` en parámetros de *lambdas*

Permite crear *lambdas* más genéricas

```
auto id = [](auto x){ return x };
auto map = [](auto c, auto f) {
  decltype(c) ret;
  std::transform(std::begin(c), std::end(c),
         std::back_inserter(ret), f);

  return ret;
};
```

Semántica de movimiento a *lambdas*

Una *lambda* podía tomar variables de su entorno por valor o por referencia, pero no podía apropiarse de ellas. Se introduce sintaxis para ello:

```
auto unwrap_and_destroy = [p = std::move(ptr)](){
                          return *p; };
```

Atributo `deprecated`

Sirve para que el compilador advierta al usuario de que está usando una entidad que va a ser eliminada en versiones futuras de una biblioteca

```
[[deprecated("g deprecated")]]
int g()
{
  return 4;
}

struct [[deprecated("s deprecated")]] s {
  int a;
};
```

std::make_unique

En C++ 11 existía `std::make_shared` para crear un objeto de tipo `std::shared_ptr`, sin embargo, había que invocar directamente el constructor de `std::unique_ptr`.

```
auto p = std::make_unique<Foo>(42);
```

Características añadidas en C++ 17

Asignaciones estructuradas

Funciona con `std::tuple`, `std::pair`, `std::array` y algunas estructuras.

```
using point3d = std::tuple<int64_t,int64_t,int64_t>;

point3d f();

auto [x, y, z] = f();
```

El atributo `nodiscard`

Al principio...

```
(void)printf("Código que has podido ver en la ETSIIT\n");
```

El atributo `nodiscard`

Muchos códigos de error pueden ser ignorados sin problema, y el compilador no suele advertir de ello. Otros no:

```
auto foo()
{
  return std::make_unique<std::thread>(f, 200);
}
```

Ejemplo

`c++14/nodiscard.cpp`

El atributo `nodiscard`

En este caso, podemos forzar una advertencia del compilador marcando `foo` con el atributo `nodiscard`.

```
[[nodiscard]]
auto foo()
{
  return std::make_unique<std::thread>(f, 200);
}
```

std::optional

Es un tipo que almacena un valor opcional de otro tipo, sin reserva de memoria dinámica.

```
auto map_opt = [](auto f, auto o)
{
  return o ? std::make_optional(f(o.value()))
           : std::nullopt;
};

auto flat_map_opt = [](auto f, auto o)
{
  return o ? f(o.value()) : std::nullopt;
};
```

std::variant

Funciona como una unión, pero permite comprobar el tipo que almacena en un momento dado.

```
std::variant<bool, string> v(true);

if(std::holds_alternative<string>(v))
 // No se ejecuta
```

Comparación: tipos de datos algebraicos

```
data Exp = Var Int          |
         | Sum Exp Exp      |
         | Product Exp Exp
```

Cualquier lenguaje que posea tipos de datos algebraicos tiene mucha más capacidad expresiva a este respecto.

Versiones paralelas de algoritmos de la biblioteca estándar

Los algoritmos definidos en la cabecera `<algorithm>` ganan un parámetro de plantilla, que indica si pueden ser paralelizados por la implementación.

```
auto sum = std::reduce(std::execution_policy::par,
                       v.begin(), v.end(), 0);
```

Bastantes cosas más

- C++11 en Wikipedia
- C++14 en Wikipedia
- C++17 en Wikipedia
- Awesome Modern C++

Fin