Lambdas en C++

Todo lo que necesitas saber sobre las **expresiones lambda** en **C++ moderno**

Desde **C++03** hasta **C++20**

Traducido por Javier Estrada

Bartłomiej Filipek

Lambdas en C++

Todo lo que necesitas saber sobre las expresiones lambda en C++ moderno, desde C++03 hasta C++20

Bartłomiej Filipek y Javier Estrada

Este libro está a la venta en http://leanpub.com/cpplambdaspanish

Esta versión se publicó en 2021-07-25

Éste es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

Índice general

Sobre el libro

Este libro muestra la historia de las expresiones lambda en C++. Aprenderás a usar esta poderosa característica paso a paso, asimilando lentamente las nuevas capacidades y mejoras que vienen con cada revisión del estándar de C++.

Comenzaremos con C++98/03 y luego pasaremos a los últimos estándares de C++.

- C++98/03 - ¿Cómo codificar sin apoyo para lambdas. ¿Cuál fue la motivación para la nueva característica moderna de C++?
- C++11 - Los primeros días. Aprenderás todos los elementos de una expresión lambda e incluso algunos trucos. Este es el capítulo más largo ya que necesitamos cubrir mucho.
- C++14 - Actualizaciones. Una vez que se adoptaron las lambdas, vimos algunas opciones para mejorarlas.
- C++17 - Más mejoras, especialmente al manejar el puntero `this` y permitir `constexpr`.
- C++20 - En esta sección veremos el estándar C++20 más reciente y novedoso.

Además, encontrarás técnicas y patrones útiles a lo largo de los capítulos para usar lambdas en tu código.

Caminar a través de la evolución de esta poderosa característica de C++ nos permite no solo aprender sobre las lambdas, sino también ver cómo ha cambiado C++ en los últimos años. En una sección verás una técnica que luego será "iterada" y actualizada en capítulos posteriores cuando haya nuevos elementos de C++ disponibles. Cuando sea posible, el libro se enlazará con otras secciones relacionadas del libro.

Orígenes del libro

La idea del contenido comenzó después de una presentación de codificación en vivo realizada por Tomasz Kamiński en nuestro Grupo de Usuarios de C++ local de Cracovia.

Tomé las ideas de la presentación (con el permiso de Tomek, por supuesto :) y luego creé dos artículos que aparecieron en bfilipek.com:

- Lambdas: desde C++11 hasta C++20, parte 1[1]
- Lambdas: desde C++11 hasta C++20, parte 2[2]

Más tarde decidí que quería ofrecer a mis lectores no solo publicaciones de blog, sino también un PDF atractivo. Leanpub proporciona una manera fácil de crear dichos PDF, por lo que fue la elección correcta copiar el contenido de los artículos y crear un libro Leanpub.

¿Por qué no ir más allá?

Después de un tiempo, decidí escribir más contenido, actualizar los ejemplos, proporcionar mejores casos de uso y patrones. ¡Y aquí tienes el libro! ¡Ahora es casi **cuatro veces** el tamaño del material inicial que está disponible en el blog!

Para quién es este libro

Este libro está dirigido a todos los desarrolladores de C++ a quienes les gusta aprender todo sobre una característica moderna de C++: las expresiones lambda.

Cómo leer este libro

Este libro tiene el orden de "historial", por lo que significa que comienza desde el fondo detrás de las lambdas y luego avanza lentamente con nuevas características y capacidades. Leer este libro de principio a fin puede ser adecuado para un desarrollador experimentado que quiera recordar los principios, ver la historia de fondo y aprender las novedades de cada estándar de C++.

Por otro lado, si eres un principiante, es mejor comenzar desde el capítulo de C++11. Ve las secciones sobre la sintaxis básica, ejemplos, cómo capturar variables. Luego, cuando estés listo, puedes omitir algunos temas avanzados y pasar al capítulo de C++14, donde aprenderás sobre las lambdas genéricas. Las primeras partes del capítulo de C++11 y C++14 son cruciales para comprender las lambdas. Una vez que obtengas los conceptos básicos, puedes leer las secciones omitidas y ver técnicas más avanzadas.

Al final del libro, en el Apéndice A hay una lista útil de "técnicas para lambdas". Puedes echar un vistazo rápido para ver si hay algo interesante y luego comenzar a leer esa sección.

[1] https://www.bfilipek.com/2019/02/lambdas-story-part1.html
[2] https://www.bfilipek.com/2019/03/lambdas-story-part2.html

Retroalimentación del lector y errata

Si detectas un error, un error tipográfico, un error gramatical o cualquier otra cosa (¡especialmente problemas lógicos!) que deba corregirse, envía tus comentarios a bartlomiej.filipek ARROBA bfilipek.com.

Aquí está la errata con la lista de correcciones:

https://www.cppstories.com/p/cpplambdaspanish/

¡Tus comentarios son importantes! Si escribes una crítica honesta, puedes ayudar con la promoción del libro y la calidad de mi trabajo posterior.

Si compraste este libro a través de Amazon, en versión impresa o Kindle, por favor deja una reseña allí.

Además, el libro tiene una página dedicada en GoodReads. Por favor comparte tus comentarios:

C++ Lambda Story @GoodReads[3]

Código de ejemplo

Puedes encontrar el código fuente de todos los ejemplos en este repositorio público independiente de Github.

github.com/fenbf/cpplambdastory-code[4]

Puedes buscar archivos individuales o descargar toda la rama:

github.com/fenbf/cpplambdastory-code/archive/main.zip[5]

Cada capítulo tiene su carpeta, por ejemplo, “Lambdas en C++11” tiene su código en “cpp11”.

Cada ejemplo tiene un número en el título. Por ejemplo:

Ex2_3: `std::function` y deducción de tipo `auto` ...

```
// código de ejemplo...
```

[3] https://www.goodreads.com/book/show/53609731-c-lambda-story
[4] https://github.com/fenbf/cpplambdastory-code
[5] https://github.com/fenbf/cpplambdastory-code/archive/main.zip

Significa que puedes ir a la carpeta del segundo capítulo - C++11 y luego buscar el tercer ejemplo. Tiene el siguiente nombre de archivo:

`chapter2_cpp11\ex2_3_std_function_and auto.cpp`.

Muchos ejemplos del libro son relativamente breves. Puedes copiar y pegar las líneas en tu compilador/entorno de desarrollo favorito y luego ejecutar el fragmento de código.

Licencia del código

El código del libro está disponible bajo la licencia Creative Commons.

Formateo y secciones especiales

Los ejemplos de código se presentan en una fuente monoespaciada, similar al siguiente ejemplo:

Para ejemplos más largos:

Título del ejemplo

```
#include <iostream>

int main() {
    const std::string text { "Hola, mundo" }
    std::cout << text << '\n';
}
```

O fragmentos más cortos (sin título y a veces con instrucciones `include`):

```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

Cuando esté disponible, también verás un enlace al compilador en línea donde puedes jugar con el código. Por ejemplo:

Título de ejemplo. Código en vivo @Wandbox

```
#include <iostream>

int main() {
    std::cout << "Hola,!";
}
```

Puedes hacer clic en el enlace en el título y luego deberá abrirse el sitio web de un compilador en línea determinado (en el caso anterior es Wandbox). Puedes compilar la muestra, ver el resultado y experimentar con el código directamente en tu navegador.

Los fragmentos de programas más largos generalmente se acortaron para presentar solo la mecánica principal.

Limitaciones de resaltado de sintaxis

La versión actual del libro puede mostrar algunas limitaciones con respecto al resaltado de sintaxis.

Por ejemplo:

- `if constexpr` - Enlace al asunto Pygments: C++ if constexpr no se reconoce (C++17) · Asunto #1136[6].
- El primer método de una clase no está resaltado - Primer método de clase no resaltado en C++ · Asunto #791[7].
- El método de plantilla no está resaltado C++ analizador léxico no reconoce la función si el tipo de retorno tiene una plantilla · Asunto #1138[8].
- Los atributos de C++ moderno a veces no se reconocen correctamente.

Otros asuntos de C++ y Pygments: Asuntos de C++ · github/pygments/pygments[9].

[6] https://github.com/pygments/pygments/issues/1136
[7] https://github.com/pygments/pygments/issues/791
[8] https://github.com/pygments/pygments/issues/1138
[9] https://github.com/pygments/pygments/issues?q=is%3Aissue+is%3Aopen+C%2B%2B

Secciones especiales

A lo largo del libro también puedes ver las siguientes secciones:

Este es un cuadro de información, con notas adicionales relacionadas con la sección actual.

Este es un cuadro de advertencia con riesgos y amenazas potenciales relacionados con un tema determinado.

> Este es un cuadro de citas. A menudo se usa en el libro para citar el estándar de C++.

Compiladores en línea

En lugar de crear proyectos locales para jugar con las muestras de código, también puedes aprovechar los compiladores en línea. Ofrecen un editor de texto básico y por lo general te permiten compilar solo un archivo fuente (el código que editas). Son convenientes si deseas jugar con ejemplos de código y verificar los resultados utilizando varios proveedores de compiladores y distintas versiones.

Por ejemplo, muchas de las muestras de código para este libro se crearon utilizando Coliru Online, Wandbox o Compiler Explorer y luego se adaptaron para el libro.

Aquí tienes una lista de algunos de los servicios útiles:

- Coliru[10] - Usa GCC 9.2.0 (a julio de 2020); ofrece compartir enlaces y un editor de texto básico, es simple pero muy efectivo.
- Wandbox[11] - Ofrece una gran cantidad de compiladores, incluidas la mayoría de las versiones de Clang y GCC, que pueden usar bibliotecas Boost; ofrece intercambio de enlaces y compilación de múltiples archivos.
- Compiler Explorer[12] - Ofrece muchos compiladores, muestra el código ensamblador

[10] http://coliru.stacked-crooked.com/
[11] https://wandbox.org/
[12] https://gcc.godbolt.org/

generado, puede ejecutar el código o incluso realizar análisis de código estático.

- CppBench[13] - Ejecuta pruebas de rendimiento simples de C++ (utilizando la biblioteca de referencia de Google).
- BuildBench[14] - Permite comparar los tiempos de construcción de dos programas C++, comparte una interfaz de usuario similar a CppBench.
- C++ Insights[15] - Una herramienta basada en Clang para la transformación de fuente a fuente. Muestra cómo el compilador ve el código al expandir las lambdas, `auto`, vínculos estructurados, deducción de plantillas y paquetes variádicos o bucles `for` basado en rango.

También hay una lista útil de compiladores en línea reunidos en este sitio web: Lista de compiladores de C++ en línea[16].

[13] http://quick-bench.com/
[14] https://build-bench.com
[15] https://cppinsights.io/
[16] https://arnemertz.github.io/online-compilers/

Sobre el autor

Bartłomiej (Bartek) Filipek es un desarrollador de software de C++ de la bella ciudad de Cracovia, al sur de Polonia. Inició su carrera profesional en 2007 y se graduó de la Universidad Jagiellonian con una Maestría en Ciencias de la Computación.

Bartek actualmente trabaja en Xara[17], donde desarrolla funciones para editores de documentos avanzados. También tiene experiencia con aplicaciones de gráficos de escritorio, desarrollo de juegos, sistemas a gran escala para la aviación, escritura de controladores de gráficos e incluso la biorretroalimentación. En el pasado, Bartek también ha enseñado programación (principalmente cursos de programación de juegos y gráficos) en universidades locales en Cracovia.

Desde 2011 Bartek ha escrito blogs regularmente en bfilipek.com[18] y últimamente en cppstories.com[19]. Inicialmente, los temas giraban en torno a la programación de gráficos, pero ahora el blog se centra en las características principales de C++. También es coorganizador del Grupo de Usuarios de C++ en Cracovia[20]. Puedes escuchar a Bartek en un episodio de @CppCast[21] donde habla sobre C++17, blogs y procesamiento de texto.

Desde octubre de 2018, Bartek ha sido un Experto de C++ para el Organismo Nacional Polaco que trabaja directamente con ISO/IEC JTC 1/SC 22 (el comité internacional ISO para la normalización del lenguaje C++). En el mismo mes, Bartek recibió su primer título de MVP para los años 2019/2020 por Microsoft.

En su tiempo libre, le encanta coleccionar y montar modelos de Lego con su pequeño hijo.

Bartek es el autor de C++17 en detalle[22].

[17] http://www.xara.com/
[18] https://www.bfilipek.com
[19] https://www.cppstories.com]
[20] https://www.meetup.com/C-User-Group-Cracow/
[21] http://cppcast.com/2018/04/bartlomiej-filipek/
[22] https://leanpub.com/cpp17indetail

Sobre el traductor

Javier Estrada es un desarrollador de software de C++ en el sur de California. Inició su carrera profesional en 1988 y se graduó del Instituto Tecnológico de Chihuahua con una Ingeniería Industrial en Electrónica.

Javier actualmente trabaja en Motorola Solutions[23], donde desarrolla software para seguridad pública en C++ y Java. En el pasado Javier también ha impartido cursos de programación en Python y Java para equipos de robótica de escuelas preparatorias regionales en el sur de California.

Javier publica en su blog Se Habla C++[24], donde trata temas generales y reseñas de presentaciones en CPPCON por distintos autores.

Javier es el traductor de C++17 - La guía completa[25] y uno de los editores principales de Referencia de C++[26]. Puedes escuchar a Javier en un par de pláticas relámpago en CPPCON: A Conversion Story: Improving from_chars and to_chars in C++17[27], e If You Build It, Will They Come?[28]

En su tiempo libre, Javier disfruta de una buena partida de ajedrez y un buen libro.

---

[23] https://motorolasolutions.com
[24] https://javierestrada.blog
[25] https://leanpub.com/cpp17es
[26] https://es.cppreference.com
[27] https://www.youtube.com/watch?v=7HB4AejLHZs
[28] https://www.youtube.com/watch?v=I8lVKve_bEk

Agradecimientos

Este libro no sería posible sin la aportación del experto de C++ **Tomasz Kamiński** (véase el perfil profesional de Tomek en Linkedin[29]).

Tomek dirigió una presentación de codificación en vivo sobre la "historia" de las lambdas en nuestro grupo de usuarios de C++ local en Cracovia:

Lambdas: desde C++11 hasta C++20 - Grupo de Usuarios de C++ en Cracovia[30]

Muchos de los ejemplos usados en este libro provienen de esa sesión.

Si bien la versión inicial del libro era relativamente corta, la versión extendida (¡100 páginas adicionales!) es el resultado de los comentarios y el aliento que recibí de **JFT (John Taylor)**. John dedicó mucho tiempo a encontrar incluso pequeñas cosas que podrían mejorarse y ampliarse.

Además, me gustaría agradecer a **Dawid Pilarski** (panicsoftware.com/about-me[31]) por sus útiles comentarios y una revisión del libro completo.

Agradecimientos adicionales para **Björn Fahller** (@playfulprogramming[32]), **Javier Estrada** (Se Habla C++[33]) y **Andreas Fertig** (andreasfertig.info[34]) por revisiones y discusiones adicionales.

Por último, pero no menos importante, recibí muchos comentarios y opiniones de los lectores del blog, Patreon Discord Server y discusiones en C++ Polska[35]. ¡Gracias a todos!

¡Con toda la ayuda de esa gente amable, la calidad del libro fue mejorando cada vez más!

[29] https://www.linkedin.com/in/tomasz-kami%C5%84ski-208572b1/
[30] https://www.meetup.com/pl-PL/C-User-Group-Cracow/events/258795519/
[31] https://blog.panicsoftware.com/about-me/
[32] https://playfulprogramming.blogspot.com/
[33] https://javierestrada.blog/
[34] https://andreasfertig.info/
[35] https://cpp-polska.pl/

Historial de revisiones

- 24 de julio de 2021 - Primera edición en línea

1. Lambdas en C++98/03

Para empezar, es bueno crear algunos antecedentes para nuestro tema principal. Para hacer esto, nos trasladaremos al pasado y veremos el código que no usa ninguna técnica de C++ moderno, lo que significa que usaremos la especificación de C++98/03.

En este capítulo aprenderás:

- Cómo pasar objetos función a los algoritmos de la biblioteca estándar de la "forma antigua".
- Las limitaciones de los tipos clase objetos función.
- Por qué los auxiliares funcionales no eran lo suficientemente buenos.
- La motivación de las lambdas para C++0x/C++11.

Objectos invocables en C++98/03

Una de las ideas fundamentales de la biblioteca estándar es que los algoritmos como `std::sort`, `std::for_each`, `std::transform` y muchos otros, pueden tomar cualquier objeto invocable y llamarlo en elementos del contenedor de entrada. Sin embargo, en C++98/03 esto solo incluía punteros a función o tipos clase con el operador de llamada a función (comúnmente denominado "*funtor*").

Como ejemplo, echemos un vistazo a una aplicación que imprime todos los elementos de un vector.

En la primera versión usaremos una función regular:

Ex1_1: Una función de impresión básica. Código en vivo @Wandbox

```
#include <algorithm>
#include <iostream>
#include <vector>

void PrintFunc(int x) {
    std::cout << x << '\n';
}

int main() {
    std::vector<int> v;
    v.push_back(1); // ¡no hay inicialización uniforme en C++03!
    v.push_back(2); // solo push_back está disponible... :)
    std::for_each(v.begin(), v.end(), PrintFunc);
}
```

El código anterior usa `std::for_each` para iterar sobre un vector (usamos C++98/03 ya que el bucle `for` basado en rango no está disponible) y luego pasa a `PrintFunc` como un objeto invocable.

Podemos convertir esta función en un tipo clase con el operador de llamada a función

Ex1_2: Un tipo objeto función de impresión básico. Código en vivo @Wandbox

```
#include <algorithm>
#include <iostream>
#include <vector>

struct Printer {
    void operator()(int x) const {
        std::cout << x << '\n';
    }
};

int main() {
    std::vector<int> v;
    v.push_back(1); // no hay lista de inicializadores
    v.push_back(2); // en C++98/03...
    std::for_each(v.begin(), v.end(), Printer());
}
```

El ejemplo define un tipo `struct` con el operador de llamada a función, `operator()`, que significa que puedes "llamar" a este objeto como una función regular:

```
Printer printer;
printer();            // llama a operator()
printer.operator()(); // llamada equivalente
```

Si bien las funciones que no son miembros suelen no tener estado[1], los tipos clase similares a funciones pueden albergar datos miembro no estáticos que permiten almacenar estado. Un ejemplo es contar el número de invocaciones de un objeto invocable en un algoritmo. Esta solución necesita mantener un contador que se actualice con cada llamada:

[1]Puedes usar variables globales o estáticas en una función regular, pero no es la mejor solución. Este enfoque dificulta el control del estado en muchos grupos de invocaciones de lambdas.

Ex1_3: Objeto función con estado. Código en vivo @Wandbox

```
#include <algorithm>
#include <iostream>
#include <vector>

struct PrinterEx {
    PrinterEx(): numCalls(0) { }

    void operator()(int x) {
        std::cout << x << '\n';
        ++numCalls;
    }

    int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    const PrinterEx vis = std::for_each(v.begin(), v.end(), PrinterEx());
    std::cout << "num calls: " << vis.numCalls << '\n';
}
```

En el ejemplo anterior, hay un dato miembro `numCalls` que se utiliza para contar el número de invocaciones del operador de llamada a función. `std::for_each` devuelve el objeto función que le pasamos, por lo que podemos tomar este objeto y obtener el dato miembro.

Como puedes predecir fácilmente, debemos obtener el siguiente resultado:

```
1
2
num calls: 2
```

También podemos "capturar" variables dentro del ámbito de la llamada. Para hacer eso, tenemos que crear un dato miembro en nuestro objeto función e inicializarlo en el constructor.

Ex1_4: Objeto función con una variable capturada. Código en vivo @Wandbox

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

struct PrinterEx {
    PrintEx(const std::string& str):
        strText(str), numCalls(0) { }

    void operator()(int x) {
        std::cout << strText << x << '\n';
        ++numCalls;
    }

    std::string strText;
    int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    const std::string introText("Elem: ");
    const PrinterEx vis = std::for_each(v.begin(), v.end(),
                                        PrinterEx(introText));
    std::cout << "num calls: " << vis.numCalls << '\n';
}
```

En esta versión, `PrinterEx` toma un parámetro adicional para inicializar un dato miembro. Entonces esta variable se usa en el operador de llamada a función y el resultado esperado es el siguiente:

```
Elem: 1
Elem: 2
num calls: 2
```

¿Qué es un "funtor"?

En algunas secciones anteriores me referí a que los tipos clase con `operator()` a veces se denominan "*funtores*". Si bien este término es útil y mucho más corto que "tipo clase objeto función", no es correcto.

Como puede verse, "*funtor*" proviene de la programación funcional y tiene un significado diferente al que se usa coloquialmente en C++.

Citando a Bartosz Milewski en funtores[2]:

> Un funtor es una correspondencia entre categorías. Dadas dos categorías, C y D, un funtor F corresponde objetos en C a objetos en D; es una función en objetos.

Es muy abstracto, pero afortunadamente, también podemos buscar una definición más simple. En el capítulo 10 de "Programación Funcional en C++"[3] Ivan Cukic "traduce" esas definiciones abstractas en una más práctica para C++:

> Una plantilla de clase `F` es un funtor si tiene una función `transform` (o `map`) definida en ella.

También, tal función `transform` debe obedecer dos reglas sobre identidad y composición.

El término "funtor" no está presente de ninguna forma en la especificación de C++ (incluso en C++98/03), por lo tanto, en el resto de este libro intentaremos evitarlo.

Recomiendo las siguientes fuentes para leer más sobre *funtores*:

- Functors, Applicatives, And Monads In Pictures - adit.io[4]
- Functors | Bartosz Milewski's Programming Cafe[5]
- What are C++ functors and their uses? - Stack Overflow[6]

[2] https://bartoszmilewski.com/2015/01/20/functors/
[3] "Programación Funcional en C++: Cómo mejorar tus programas de C++ usando técnicas funcionales 1ra Edición" @Amazon
[4] https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
[5] https://bartoszmilewski.com/2015/01/20/functors/
[6] https://stackoverflow.com/questions/356950/what-are-c-functors-and-their-uses

- Funtor - Wikipedia[7]

Problemas con los tipos clase objeto función

Como puedes ver, crear tipos clase con el operador de llamada a función es muy poderoso. Tienes control total y puedes diseñarlos como quieras.

Sin embargo, en C++98/03 el problema era que tenías que definir un tipo objeto función en un lugar diferente al de la invocación del algoritmo. Esto podría significar que el invocable podría estar decenas o cientos de líneas antes o después en el archivo fuente, o incluso en una unidad de compilación diferente.

Como solución potencial, es posible que hayas intentado escribir clases locales, ya que C++ siempre ha admitido esa sintaxis, pero eso no funcionó con las plantillas.

Observa este código:

Un tipo objeto función local

```
int main() {
    struct LocalPrinter {
        void operator()(int x) const {
            std::cout << x << '\n';
        }
    };

    std::vector<int> v(10, 1);
    std::for_each(v.begin(), v.end(), LocalPrinter());
}
```

Trata de compilarlo con `-std=c++98` y verás el siguiente error en GCC:

```
error: template argument for
'template<class _IIter, class _Funct> _Funct
std::for_each(_IIter, _IIter, _Funct)'
uses local type 'main()::LocalPrinter'
```

[7] https://es.wikipedia.org/wiki/Funtor

Como puede verse, en C++98/03 no se podía crear una instancia de una plantilla con un tipo local.

Los programadores de C++ comprendieron rápidamente esas limitaciones y encontraron formas de solucionar los problemas con C++98/03. Una solución fue preparar un conjunto de auxiliares. Revisémoslos en la siguiente sección.

Composición con auxiliares funcionales

¿Qué tal tener algunos auxiliares y objetos función predefinidos?

Si revisas el archivo de encabezado `<functional>` de la biblioteca estándar, encontrarás muchos tipos y funciones que se pueden usar inmediatamente con los algoritmos estándar.

Por ejemplo:

- `std::plus<T>()` - Toma dos argumentos y devuelve su suma.
- `std::minus<T>()` - Toma dos argumentos y devuelve su diferencia.
- `std::less<T>()` - Toma dos argumentos y devuelve si el primero es menor que el segundo.
- `std::greater_equal<T>()` - Toma dos argumentos y devuelve si el primero es mayor o igual que el segundo.
- `std::bind1st` - Crea un objeto invocable con el primer argumento fijo a un valor dado.
- `std::bind2nd` - Crea un objeto invocable con el segundo argumento fijo a un valor dado.
- `std::mem_fun` - Crea un objeto envolvente de una función miembro.
- y muchos más.

Escribamos algo de código que se beneficie de los auxiliares:

Ex1_5: Usar los viejos auxiliares funcionales de C++98/03. Código en vivo @Wandbox

```
#include <algorithm>
#include <functional>
#include <vector>

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    // .. push back hasta 9...
    const size_t smaller5 = std::count_if(v.begin(), v.end(),
                                          std::bind2nd(std::less<int>(), 5));

    return smaller5;
}
```

El ejemplo usa `std::less` y fija su segundo argumento usando `std::bind2nd`. Toda esta "composición" se pasa a `count_if`[8]. Como probablemente puedes adivinar, el código se expande en una función que realiza una comparación simple:

```
return x < 5;
```

Si deseas más auxiliares listos para usar, también puedes ver la biblioteca Boost. Por ejemplo, `boost::bind`.

Desafortunadamente, el problema principal con este enfoque es la complejidad y la sintaxis difícil de aprender. Por ejemplo, escribir código que componga dos o más funciones no es natural. Echa un vistazo a continuación:

[8] `bind1st`, `bind2nd` y otros auxiliares funcionales quedaron obsoletos en C++11 y se eliminaron en C++17. El código de este capítulo los usa solo para ilustrar problemas de C++98/03. Utiliza otras alternativas modernas en tus proyectos. Véase el capítulo de C++14 para más información.

Ex1_6: Componer auxiliares funcionales. Código en vivo @Wandbox

```
#include <algorithm>
#include <functional>
#include <vector>

int main() {
    using std::placeholders::_1;

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    // push_back hasta 9...
    const size_t val = std::count_if(v.begin(), v.end(),
                                     std::bind(std::logical_and<bool>(),
                                     std::bind(std::greater<int>(),_1, 2),
                                     std::bind(std::less_equal<int>(),_1,6)));
    return val;
}
```

La composición usa `std::bind` (de C++11, así que hicimos un poco de trampa, no es C++98/03) con `std::greater` y `std::less_equal` conectados con `std::logical_and`. Además, el código usa `_1`, que es un marcador de posición para el primer argumento de entrada.

Si bien el código anterior funciona y puedes definirlo localmente, probablemente estés de acuerdo en que es una sintaxis complicada y no natural. Sin mencionar que esta composición representa solo una condición simple:

```
return x > 2 && x <= 6;
```

¿Hay algo mejor y más sencillo de usar?

Motivación para una nueva característica

Como puedes ver, en C++98/03 había varias formas de declarar y pasar un objeto invocable a los algoritmos y las utilerías de la biblioteca estándar. Sin embargo, todas esas opciones

eran un poco limitadas. Por ejemplo, no se podía declarar un tipo objeto función local, o era complicado componer una función con objetos auxiliares funcionales.

¡Afortunadamente con C++11 finalmente vimos muchas mejoras!

En primer lugar, el comité internacional ISO para la normalización del lenguaje C++ eliminó la limitación de la instanciación de plantillas con un tipo local. Desde C++11, puedes escribir tipos clase con el operador de llamada a función localmente en el lugar donde los necesites.

Es más, C++11 también dio vida a otra idea: ¿qué pasa si tenemos una sintaxis corta y luego el compilador puede "expandirla" en una definición de tipo objeto función local?

¡Y ese fue el nacimiento de las "expresiones lambda"!

Si vemos N3337[9], el borrador final de C++11, podemos ver una sección separada para lambdas: [expr.prim.lambda][10].

Echemos un vistazo a esta nueva característica en el próximo capítulo.

[9] https://timsong-cpp.github.io/cppwp/n3337/
[10] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda

2. Lambdas en C++11

¡Hurra! El comité internacional ISO para la normalización del lenguaje C++ escuchó las opiniones de los desarrolladores y desde C++11 tenemos las expresiones lambda.

Las lambdas se convirtieron rápidamente en una de las características más reconocibles de C++ moderno.

Puedes leer la especificación completa que se encuentra en N3337[1]: el borrador final de C++11.

Y la sección separada para lambdas: [expr.prim.lambda][2].

Creo que el comité agregó las lambdas al lenguaje de una manera inteligente. Incorporan una nueva sintaxis, pero luego el compilador las "expande" en un tipo objeto función "oculto" sin nombre. De esta manera tenemos todas las ventajas (y desventajas) del lenguaje real fuertemente tipado, y es relativamente fácil razonar sobre el código.

En este capítulo aprenderás:

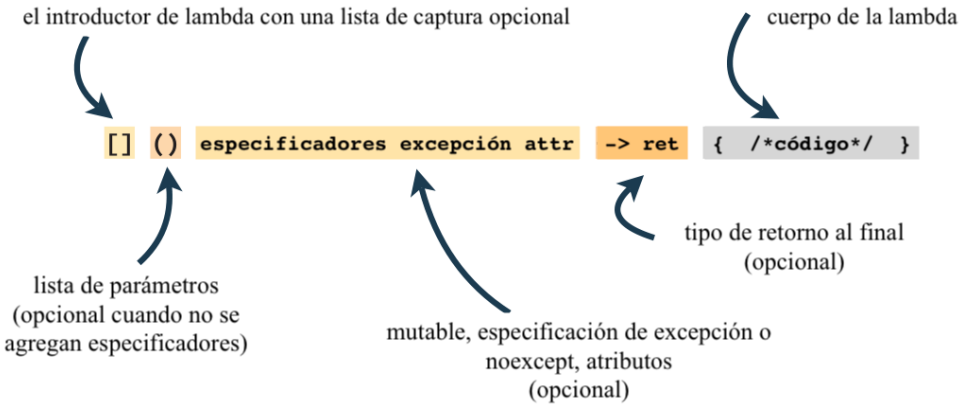
- La sintaxis básica de las lambdas.
- Cómo capturar variables.
- Cómo capturar datos miembro no estáticos de una clase.
- El tipo de retorno de una lambda.
- Lo que es un objeto cierre.
- Cómo una lambda puede convertirse en un puntero a función y usarse con una API estilo C.
- Lo que es el acrónimo EFII y por qué es útil.
- Cómo heredar de una expresión lambda.

¡Vamos!

[1] https://timsong-cpp.github.io/cppwp/n3337/
[2] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda

La sintaxis de una expresión lambda

A continuación puedes encontrar un diagrama que ilustra la sintaxis de las lambdas en C++11:



Sintaxis de las lambdas en C++11

Veamos algunos ejemplos para crear intuición.

Algunos ejemplos de las expresiones lambda

```
// 1. La lambda más simple:
[]{};
```

En el primer ejemplo, puedes ver una expresión lambda "mínima". Solo necesita la sección `[]` (el introductor lambda) y luego la parte vacía `{}` para el cuerpo de la función. La lista de argumentos - `()` - es opcional y no es necesaria en este caso.

```
// 2. Con dos parámetros:
[](float f, int a) { return a * f; };
[](int a, int b) { return a < b; };
```

En el segundo ejemplo, probablemente uno de los más comunes, puedes ver que los argumentos se pasan a la sección `()` al igual que para una función normal. El tipo de retorno no es necesario, ya que el compilador lo deducirá automáticamente.

```
// 3. Tipo de retorno al final:
[](MyClass t) -> int { auto a = t.compute(); print(a); return a; };
```

En el ejemplo anterior, establecemos explícitamente un tipo de retorno. El tipo de retorno al final también está disponible para la declaración de funciones regulares desde C++11.

```
// 4. Especificadores adicionales:
[x](int a, int b) mutable { ++x; return a < b; };
[](float param) noexcept { return param*param; };
[x](int a, int b) mutable noexcept { ++x; return a < b; };
```

El último ejemplo muestra que puedes usar otros especificadores antes del cuerpo de la lambda. En el código usamos `mutable` (para poder cambiar la variable capturada) y también `noexcept`. La tercera lambda usa `mutable` y `noexcept` y tienen que aparecer en ese orden (no puedes escribir `noexcept mutable` ya que el compilador lo rechazaría). Si bien la parte `()` es opcional, si deseas aplicar `mutable` o `noexcept`, entonces `()` debe estar en la expresión:

```
// 5. () opcional
[x] { std::cout << x; }; // no se necesita ()
[x] mutable { ++x; };    // ¡no compila!
[x]() mutable { ++x; };  // está bien - se requiere () antes de mutable
[] noexcept { };         // ¡no compila!
[]() noexcept { };       // está bien
```

El mismo patrón se aplica a otros especificadores que se pueden aplicar en las lambdas, como `constexpr` o `consteval` en C++17 y C++20, respectivamente.

Después de los ejemplos básicos, ahora podemos intentar comprender cómo funciona y aprender todas las posibilidades de las expresiones lambda.

Definiciones básicas

Antes de continuar, es útil traer algunas definiciones básicas del estándar de C++:

De [expr.prim.lambda#2][3]:

> La evaluación de una expresión lambda da como resultado un valor temporal. Este temporal se llama **objeto cierre**.

Como nota al margen, una expresión lambda es un pr-valor que es un "r-valor puro". Este tipo de expresiones generalmente producen inicializaciones y aparecen en el lado derecho de la asignación (o en una declaración de retorno). Leer más en la referencia de C++[4].

Y otra definición de [expr.prim.lambda#3][5]:

> El tipo de la expresión lambda (que también es el tipo del objeto cierre) es un tipo clase no unión único, sin nombre, llamado **tipo cierre**.

Expansión del compilador

De las definiciones anteriores, podemos entender que el compilador genera algún tipo cierre único a partir de una expresión lambda. Entonces podemos tener una instancia de este tipo a través del objeto cierre.

Aquí hay un ejemplo básico que muestra cómo escribir una expresión lambda y pasarla a `std::for_each`. A modo de comparación, el código también ilustra el tipo objeto función correspondiente generado por el compilador:

[3] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#2
[4] https://es.cppreference.com/w/cpp/language/value_category
[5] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#3

Ex2_1: Una lambda y un tipo objeto función correspondiente. Código en vivo @WandBox

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    struct {
        void operator()(int x) const {
            std::cout << x << '\n';
        }
    } someInstance;

    const std::vector<int> v { 1, 2, 3 };
    std::for_each(v.cbegin(), v.cend(), someInstance);
    std::for_each(v.cbegin(), v.cend(), [] (int x) {
            std::cout << x << '\n';
        }
    );
}
```

En el ejemplo, el compilador transforma ...

```
[](int x) { std::cout << x << '\n'; }
```

... en un objeto función anónimo, que en una forma simplificada puede verse de la siguiente manera:

```
struct {
    void operator()(int x) const {
        std::cout << x << '\n';
    }
} someInstance;
```

El proceso de traducción o "expansión" se puede ver fácilmente en C++ Insights[6], una herramienta en línea que toma código C++ válido y luego produce una versión del código

[6] https://cppinsights.io/

fuente que el compilador genera: similar a los objetos función anónimos para las lambdas, la instanciación de plantillas y muchas otras características de C++.

En las siguientes secciones, profundizaremos más en las partes individuales de la expresión lambda.

El tipo de una expresión lambda

Dado que el compilador genera un nombre único para cada lambda (el tipo cierre), no hay forma de "deletrearlo" por adelantado.

Es por eso que tienes que usar `auto` (o `decltype`) para deducir el tipo.

```
auto myLambda = [](int a) -> double { return 2.0 * a; };
```

Además, si tienes dos lambdas que se ven iguales:

```
auto firstLam  = [](int x) { return x * 2; };
auto secondLam = [](int x) { return x * 2; };
```

¡Sus tipos son diferentes incluso si el "código subyacente" es el mismo! El compilador debe declarar dos tipos únicos sin nombre para cada lambda.

Podemos probar esta propiedad con el siguiente código:

Ex2_1: Tipos distintos, mismo código. Código en vivo @Wandbox

```
#include <type_traits>

int main() {
    const auto oneLam = [](int x) noexcept { return x * 2; };
    const auto twoLam = [](int x) noexcept { return x * 2; };
    static_assert(!std::is_same<decltype(oneLam), decltype(twoLam)>::value,
                  "must be different!");
}
```

El ejemplo anterior verifica si los tipos de cierre para `oneLam` y `twoLam` no son iguales.

En C++17 podemos usar `static_assert` sin mensaje y también plantillas de variable auxiliares para los rasgos de tipo `is_same_v`:

```
static_assert(std::is_same_v<double, decltype(baz(10))>);
```

Sin embargo, aunque no conoces el nombre exacto, puedes deletrear la signatura de la lambda y luego almacenarla en `std::function`. En general, lo que no se puede hacer con una lambda definida como `auto` se puede hacer si la lambda se "expresa" a través del tipo `std::function<>`. Por ejemplo, la lambda anterior tiene una signatura de `double(int)` ya que toma un `int` como parámetro de entrada y devuelve `double`. Entonces podemos crear un objeto `std::function` de la siguiente manera:

```
std::function<double(int)> myFunc = [](int a) -> double { return 2.0 * a; };
```

`std::function` es un objeto pesado porque necesita manejar todos los objetos invocables. Para hacer eso, requiere mecánicas internas avanzadas como manipulación de tipos o incluso asignación de memoria dinámica. Podemos comprobar su tamaño en un experimento sencillo:

Ex2_3: **`std::function`** y deducción de tipo **`auto`**. Código en vivo **@Wandbox**

```
#include <functional>
#include <iostream>

int main() {
    const auto myLambda = [](int a) noexcept -> double {
        return 2.0 * a;
    };

    const std::function<double(int)> myFunc =
        [](int a) noexcept -> double {
            return 2.0 * a;
        };

    std::cout << "sizeof(myLambda) is " << sizeof(myLambda) << '\n';
    std::cout << "sizeof(myFunc) is " << sizeof(myFunc) << '\n';

    return myLambda(10) == myFunc(10);
}
```

En GCC el código imprimirá:

```
sizeof(myLambda) is 1
sizeof(myFunc) is 32
```

Ya que `myLambda` es solo una lambda sin estado, también es una clase vacía, sin ningún dato miembro, por lo que su tamaño mínimo es de solo un byte. Por otro lado, la versión `std::function` es mucho más grande: 32 bytes. Es por eso que si puedes, confía en la deducción de tipo `auto` para obtener el número de objetos cierre más pequeño posible.

Cuando hablamos de `std::function`, también es importante mencionar que este tipo no admite cierres solo movibles. Puedes leer más sobre este asunto en el capítulo de C++14 sobre tipos movibles.

Constructores y copia

La sección está disponible solo en la versión completa del libro.

El operador de llamada a función

El código que pones en el cuerpo de la lambda se "traduce" al código en el `operator()` del tipo cierre correspondiente.

Por defecto, en C++11 es una función miembro `const inline`. Por ejemplo:

```
auto lam = [](double param) { /* hacer algo */ };
```

Se expande en algo similar a:

```
struct __anonymousLambda {
    inline void operator()(double param) const { /* hacer algo */ }
};
```

Analicemos las consecuencias de este enfoque y cómo podemos modificar la declaración del operador de llamada a función resultante.

Sobrecarga

Una cosa que vale la pena mencionar es que cuando defines una lambda, no hay forma de crear lambdas "sobrecargadas" tomando argumentos distintos. Por ejemplo:

```
// ¡no compila!
auto lam = [](double param) { /* hacer algo */ };
auto lam = [](int param) { /* hacer algo */ };
```

Arriba, el código no se compilará ya que el compilador no puede traducir esas dos lambdas en un solo objeto función. Además, no puedes redefinir la misma variable. Por otro lado, es posible crear un tipo objeto función que tenga dos operadores de llamada a función:

```
struct MyFunctionObject {
    inline void operator()(double param) const { /* hacer algo */ }
    inline void operator()(int param) const { /* hacer algo */ }
};
```

`MyFunctionObject` ahora puede funcionar con argumentos `double` e `int`. Si deseas un comportamiento similar para las lambdas, puedes ver la sección sobre la herencia de lambdas en este capítulo y también sobre el patrón de sobrecarga del capítulo de C++17.

Atributos

La sintaxis para lambdas permite usar los atributos de C++11 en forma de `[[attr_name]]`. Sin embargo, si aplicas un atributo a una lambda, se aplica al tipo del operador de llamada a función y no al operador en sí. Es por eso que actualmente (e incluso en C++20) no hay atributos que tengan sentido poner en una lambda. La mayoría de los compiladores incluso informan de un error. Si tomamos un atributo de C++17 e intentamos usarlo con la expresión:

```
auto myLambda = [](int a) [[nodiscard]] { return a * a; };
```

Esto genera el siguiente error en Clang (véase el código en vivo @Wandbox[7]):

```
error: 'nodiscard' attribute cannot be applied to types
```

Si bien en teoría la sintaxis para lambdas está preparada, por el momento no existen atributos aplicables.

[7] https://wandbox.org/permlink/3zfzL1NNpPXXgLOx

Otros modificadores

Tocamos brevemente este tema en la sección de sintaxis, pero no está limitado a una declaración por defecto del operador de llamada a función para un tipo cierre. En C++11 puedes agregar `mutable` o una especificación de excepción.

Si es posible, ejemplos más largos de este libro intentan marcar el objeto cierre con `const` y también hacer la lambda `noexcept`.

Puedes usar esas palabras clave especificando `mutable` y `noexcept` después de la cláusula de declaración de parámetros:

```
auto myLambda = [](int a) mutable noexcept { /* hacer algo */ }
```

El compilador expandirá este código en:

```
struct __anonymousLambda {
    inline void operator()(double param) noexcept { /* hacer algo */ }
};
```

Ten en cuenta que la palabra clave `const` desapareció y el operador de llamada a función ahora puede cambiar los datos miembro de la lambda.

¿Pero qué datos miembro? ¿Cómo podemos declarar un dato miembro de una lambda? Veamos la siguiente sección sobre "captura" de variables:

Capturas

La sección está disponible solo en la versión completa del libro.

Código generado

A lo largo de este libro, muestro un posible código generado por el compilador como un tipo `struct` para definir un tipo de clase cierre. Sin embargo, esto es solo una simplificación, un modelo mental, y dentro del compilador podría ser diferente.

Por ejemplo, en Clang el árbol de sintaxis abstracto (AST por sus siglas en inglés) usa `class` para representar un cierre. El operador de llamada a función se define como `public` mientras que los datos miembros son `private`.

Es por eso que no puedes escribir:

```
int x = 0;
auto lam = [x]() { std::cout << x; };
lam.x = 10; // ??
```

En GCC (o de manera similar en Clang) obtendrás:

```
error: 'struct main()::<lambda()>' has no member named 'x'
```

Por otro lado, tenemos una parte esencial de la especificación que menciona que las variables capturadas se inicializan directamente, lo cual es imposible para miembros privados (para nuestras clases regulares en código). Esto significa que aquí los compiladores pueden hacer un poco de "magia" y crear un código más eficiente (no hay necesidad de copiar variables o incluso moverlas).

Puedes leer más sobre los componentes internos de las lambdas en una excelente publicación de blog de Andreas Fertig (el creador de C++ Insights): Las lambdas de C++ bajo las sábanas - Part 2: Capturas, capturas, capturas[8].

¿Capturar todo o explícitamente?

Aunque especificar `[=]` o `[&]` puede ser conveniente, ya que se capturan todas las variables con duración de almacenamiento automática, es más claro capturar una variable explícitamente. De esa forma el compilador puede advertirte sobre efectos no deseados (por ejemplo, consulta las notas sobre variables globales y estáticas).

También puedes leer más en el artículo 31 de "Effective Modern C++"[9] por Scott Meyers: "Avoid default capture modes."

[8] https://andreasfertig.blog/2020/11/under-the-covers-of-cpp-lambdas-part-2-captures-captures-captures/

[9] "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14" primera edición por Scott Meyers, 2014

la palabra clave `mutable`

Por defecto, `operator()` del tipo cierre está marcado como `const`, y no puedes modificar las variables capturadas dentro del cuerpo de la lambda.

Si deseas cambiar este comportamiento, debes agregar la palabra clave `mutable` después de la lista de parámetros. Esta sintaxis elimina efectivamente a `const` de la declaración del operador de llamada a función en el tipo cierre. Si tienes una expresión lambda simple con `mutable`:

```
int x = 1;
auto foo = [x]() mutable { ++x; };
```

Se "expandirá" en el siguiente objeto función:

```
struct __lambda_x1 {
    void operator()() { ++x; }
    int x;
};
```

Como puedes ver, el operador de llamada a función puede cambiar el valor de los datos miembro.

Ex2_5: Captura de dos variables por copia y mutable. Código en vivo @Wandbox

```
#include <iostream>

int main() {
    const auto print = [](const char* str, int x, int y) {
        std::cout << str << ": " << x << " " << y << '\n';
    };
    int x = 1, y = 1;
    print("in main()", x, y);
    auto foo = [x, y, &print]() mutable {
        ++x;
        ++y;
        print("in foo()", x, y);
    };
    foo();
```

```
    print("in main()", x, y);
}
```

Salida:

```
in main(): 1 1
in foo(): 2 2
in main(): 1 1
```

En el ejemplo anterior, podemos cambiar los valores de `x` e `y`. Dado que esas son solo las copias de `x` e `y` del ámbito adjunto, no vemos sus nuevos valores después de que se invoca a `foo`.

Por otro lado, si capturas por referencia no necesitas aplicar `mutable` a la lambda para modificar el valor. Esto se debe a que los datos miembro capturados son referencias, lo que significa que de ningún modo puedes volver a vincularlos a un nuevo objeto, pero puedes cambiar los valores referenciados.

```
int x = 1;
std::cout << x << '\n';
const auto foo = [&x]() noexcept { ++x; };
foo();
std::cout << x << '\n';
```

En el ejemplo anterior, la lambda no se especifica con `mutable` pero puedes cambiar el valor al que se hace referencia.

Una cosa importante a tener en cuenta es que cuando aplicas `mutable`, entonces no puedes marcar el objeto cierre resultante con `const`, ya que te impide invocar la lambda.

```
int x = 10;
const auto lam = [x]() mutable { ++x; }
lam(); // ¡no compila!
```

La última línea no se compilará ya que no podemos llamar a una función miembro que no sea `const` en un objeto `const`.

Contador de invocación - Un ejemplo de variables capturadas

Antes de pasar a algunos temas de captura más complicados, podemos hacer una pequeña pausa y centrarnos en un ejemplo más práctico.

Las expresiones lambda son útiles cuando deseas utilizar algún algoritmo existente de la biblioteca estándar y modificar el comportamiento predeterminado. Por ejemplo, para `std::sort` puedes escribir su función de comparación.

Pero podemos ir más allá y mejorar el comparador con un contador de invocaciones. Echemos un vistazo:

Ex2_6: Contador de invocación. Código en vivo @Compiler Explorer

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec { 0, 5, 2, 9, 7, 6, 1, 3, 4, 8 };

    size_t compCounter = 0;
    std::sort(vec.begin(), vec.end(),
        [&compCounter](int a, int b) noexcept {
            ++compCounter;
            return a < b;
        }
    );

    std::cout << "number of comparisons: " << compCounter << '\n';

    for (const auto& v : vec)
        std::cout << v << ", ";
}
```

El comparador proporcionado en el ejemplo funciona de la misma manera que el predeterminado, retorna si `a` es menor que `b`, por lo que usamos el orden natural de los números de menor a mayor. Sin embargo, la lambda pasada a `std::sort` también captura una variable local `compCounter`. Luego, la variable se usa para contar todas las invocaciones de este comparador desde el algoritmo de ordenamiento.

Captura de variables globales

Si tienes una variable global y usas `[=]` en su lambda, podrías pensar que su objeto global también se captura por valor ... pero no lo es. Ve el código:

Ex2_7: Captura de variables globales. Código en vivo @Wandbox

```
#include <iostream>

int global = 10;

int main() {
    std::cout << global << '\n';
    auto foo = [=]() mutable noexcept { ++global; };
    foo();
    std::cout << global << '\n';
    const auto increaseGlobal = []() noexcept { ++global; };
    increaseGlobal();
    std::cout << global << '\n';
    const auto moreIncreaseGlobal = [global]() noexcept { ++global; };
    moreIncreaseGlobal();
    std::cout << global << '\n';
}
```

El ejemplo anterior define `global` y luego la usa con varias lambdas definidas en la función `main()`. Si ejecutas el código, no importa la forma en que captures, siempre apuntará al objeto global y no se crearán copias locales.

Es porque solo se pueden capturar variables con duración de almacenamiento automática. GCC incluso puede emitir la siguiente advertencia:

```
warning: capture of variable 'global' with non-automatic
         storage duration
```

Esta advertencia aparecerá solo si capturas explícitamente una variable global, por lo que si usas `[=]` el compilador no te ayudará.

El compilador Clang es incluso más útil, ya que genera **un error**:

```
error: 'global' cannot be captured because it does not have
       automatic storage duration
```

Véase el ejemplo en vivo con Clang @Wandbox[10].

Captura de variables estáticas

De manera similar a la captura de variables globales, tendrás los mismos problemas con los objetos estáticos:

Ex2_8: Captura de variables estáticas. Código en vivo @Wandbox

```
#include <iostream>

void bar() {
    static int static_int = 10;
    std::cout << static_int << '\n';
    auto foo = [=]() mutable noexcept{ ++static_int; };
    foo();
    std::cout << static_int << '\n';
    const auto increase = []() noexcept { ++static_int; };
    increase();
    std::cout << static_int << '\n';
    const auto moreIncrease = [static_int]() noexcept { ++static_int; };
    moreIncrease();
    std::cout << static_int << '\n';
}

int main() {
   bar();
}
```

Esta vez intentamos capturar una variable estática y luego cambiar su valor, pero como no tiene una duración de almacenamiento automática, el compilador no puede hacerlo.

Salida:

[10] https://wandbox.org/permlink/4V91bkuz8NvHrDDA

```
10
11
12
13
```

GCC emite una advertencia cuando capturas la variable por nombre `[static_int]` y Clang emite un error.

Captura de datos miembro y el puntero `this`

Las cosas se complican un poco más cuando estás en una función miembro de una clase y quieres capturar un dato miembro. Dado que todos los datos miembro no estáticos están relacionados con el puntero `this`, el dato miembro también debe almacenarse en algún lugar.

Echemos un vistazo:

Ex2_9: Error al capturar un dato miembro. Código en vivo @Wandbox

```
#include <iostream>

struct Baz {
    void foo() {
        const auto lam = [s]() { std::cout << s; };
        lam();
    }

    std::string s;
};

int main() {
    Baz b;
    b.foo();
}
```

El código intenta capturar a `s`, que es un dato miembro, pero el compilador emitirá el siguiente mensaje de error:

```
In member function 'void Baz::foo()':
error: capture of non-variable 'Baz::s'
error: 'this' was not captured for this lambda function
```

Para resolver este problema, debes capturar el puntero `this` y entonces tendrás acceso a los datos miembro.

Podemos actualizar el código a:

```
struct Baz {
    void foo() {
        const auto lam = [this]() { std::cout << s; };
        lam();
    }

    std::string s;
};
```

Ahora no hay errores del compilador.

También puedes usar `[=]` o `[&]` para capturar `this` (¡ambos tienen el mismo efecto en C++11/14!).

Ten en cuenta que capturamos `this` por valor ... a un puntero. Es por eso que tienes acceso al dato miembro inicial, no a su copia.

En C++11 (e incluso en C++14) no puedes escribir:

```
auto lam = [*this]() { std::cout << s; };`
```

El código no se compilará en C++11/14; sin embargo, está permitido en C++17.

Si usas tus lambdas en el contexto de una sola función, entonces capturar `this` estará bien. Pero, ¿qué hay de los casos más complicados?

¿Sabes qué pasará con el siguiente código?

Ex2_10: Devolver una lambda desde una función

```
#include <functional>
#include <iostream>

struct Baz {
    std::function<void()> foo() {
        return [=] { std::cout << s << '\n'; };
    }

    std::string s;
};

int main() {
   auto f1 = Baz{"abc"}.foo();
   auto f2 = Baz{"xyz"}.foo();
   f1();
   f2();
}
```

El código declara un objeto `Baz` y luego invoca a `foo()`. Ten en cuenta que `foo()` devuelve una lambda (almacenada en `std::function`) que captura un miembro de la clase[11].

Dado que usamos objetos temporales, no podemos estar seguros de lo que sucederá cuando llames a `f1` y `f2`. Este es un problema de referencia pendiente y genera un comportamiento indefinido.

Similarmente para:

```
struct Bar {
    std::string const& foo() const { return s; };
    std::string s;
};
auto&& f1 = Bar{"abc"}.foo(); // una referencia pendiente
```

Juega con el código @Wandbox[12].

Si indicas la captura explícitamente (`[s]`) obtendrás un error del compilador.

[11] `std::function` se requiere en C++11 ya que no hay deducción de tipo de retorno para funciones. Esta limitación se elimina en C++14.

[12] https://wandbox.org/permlink/FOgbNGoQHOmepBgY

```
std::function<void()> foo() {
    return [s] { std::cout << s << '\n'; };
} // error: ¡'this' no se capturó!
```

Con todo, capturar `this` puede resultar complicado cuando una lambda puede vivir más allá del objeto en sí. Esto puede suceder cuando utilizas llamadas asíncronas o múltiples hilos.

Volveremos a ese tema en el capítulo de C++17. Véase "Ejecución concurrente mediante lambdas" en el capítulo de C++17 en la página 106.

Objetos solo movibles

La sección está disponible solo en la versión completa del libro.

Conservación de `const`

Si capturas una variable `const`, entonces la constancia se conserva:

Ex2_11: Conservación de `const`. Código en vivo @Wandbox

```
#include <iostream>
#include <type_traits>

int main() {
    const int x = 10;
    auto foo = [x] () mutable {
        std::cout << std::is_const<decltype(x)>::value << '\n';
        x = 11;
    };
    foo();
}
```

El código anterior no se compila ya que la variable capturada es constante. Aquí tienes un posible objeto función generado para este ejemplo:

```
struct __lambda_x {
    void operator()() { x = 11; /*error!*/ }
    const int x;
};
```

También puedes jugar con este código en @CppInsight[13].

Captura de un paquete de parámetros

Para cerrar nuestra discusión sobre la cláusula de captura, debemos mencionar que también puedes aprovechar las capturas con plantillas variádicas. El compilador expande el paquete en una lista de datos miembro no estáticos que pueden ser útiles si deseas usar una lambda en un código con plantilla. Por ejemplo, aquí hay una muestra de código que experimenta con las capturas:

Ex2_12: Captura de un paquete de parámetros. Código en vivo @Wandbox

```
#include <iostream>
#include <tuple>

template<class... Args>
void captureTest(Args... args) {
    const auto lambda = [args...] {
        const auto tup = std::make_tuple(args...);
        std::cout << "tuple size: " <<
                     std::tuple_size<decltype(tup)>::value << '\n';
        std::cout << "tuple 1st:  " << std::get<0>(tup) << '\n';
    };
    lambda(); // call it
}

int main() {
    captureTest(1, 2, 3, 4);
    captureTest("Hello world", 10.0f);
}
```

Después de ejecutar el código, obtendremos el siguiente resultado:

[13] https://cppinsights.io/s/7b2f8b10

```
tuple size: 4
tuple 1st:  1
tuple size: 2
tuple 1st:  Hello world
```

Este código algo experimental muestra que puedes capturar un paquete de parámetros variádico por valor (por referencia también es posible) y luego el paquete se "almacena" en un objeto tupla. Luego llamamos a algunas funciones auxiliares en la tupla para acceder a sus datos y propiedades.

También puedes usar C++ Insights para ver cómo el compilador genera el código y expande las plantillas, los paquetes de parámetros y las lambdas en código. Ve este ejemplo aquí: @C++Insights[14].

Véase el capítulo de C++14 donde es posible capturar un tipo solo movibles y también el capítulo de C++20 para mejoras con el paquete de parámetros variádico.

Tipo de retorno

En la mayoría de los casos, incluso en C++11, puedes omitir el tipo de retorno de la lambda y luego el compilador deducirá el nombre del tipo por ti.

Como nota al margen: Inicialmente, la deducción del tipo de retorno se restringió a las lambdas con cuerpos que contenían una única declaración de retorno. Sin embargo, esta restricción se eliminó rápidamente ya que no hubo problemas para implementar una versión más conveniente.

Véase Informes de defectos del lenguaje principal estándar de C++ y problemas aceptados[15].

En resumen, desde C++11 el compilador ha podido deducir el tipo de retorno siempre que todas sus declaraciones de retorno sean del mismo tipo.

Del informe de defectos podemos leer lo siguiente [16]:

[14] https://cppinsights.io/s/19d3a45d
[15] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#975
[16] Gracias a Tomek Kamiński por encontrar el enlace correcto.

> ...Si una *expresión-lambda* no incluye un *tipo-de-retorno-al-final*, es como si el *tipo-de-retorno-al-final* denotara el siguiente tipo:
>
> - si no hay declaraciones de retorno en la instrucción compuesta, o todas las declaraciones de retorno devuelven una expresión de tipo `void` o ninguna expresión o lista de inicializadores entre llaves, el tipo `void`;
> - de lo contrario, si todas las instrucciones de retorno devuelven una expresión y los tipos de expresiones devueltas después de la conversión de l-valor a r-valor (7.3.2 [conv.lval]), conversión de *array* a puntero (7.3.3 [conv.array]), y conversión de función a puntero (7.3.4 [conv.func]) son las mismas, ese tipo común;
> - de lo contrario, el programa está mal formado.

Ex2_13: Deducción del tipo de retorno. Código en vivo @Wandbox

```
#include <type_traits>

int main() {
    const auto baz = [](int x) noexcept {
        if (x < 20)
            return x * 1.1;
        else
            return x * 2.1;
    };
    static_assert(std::is_same<double, decltype(baz(10))>::value,
                  "has to be the same!");
}
```

En la lambda anterior tenemos dos instrucciones `return`, pero ambas apuntan a `double`, por lo que el compilador puede deducir el tipo.

En C++14 el tipo de retorno de una lambda se actualizará para adaptarse a las reglas de deducción de tipo `auto` para las funciones regulares. Véase "Deducción del tipo de retorno" en la página 56. Esto resulta en una definición mucho más simple.

Sintaxis del tipo de retorno al final

La sección está disponible solo en la versión completa del libro.

Conversión a un puntero a función

Si tu lambda no captura ninguna variable, entonces el compilador puede convertirla en un puntero a función regular. Véase la siguiente descripción del estándar expr.prim.lambda#6[17]:

> El tipo cierre para una expresión lambda sin captura de lambda tiene una función de conversión pública, no virtual, no explícita, `const` a puntero a función que tiene el mismo parámetro y tipos de retorno que el operador de llamada a función del tipo cierre. El valor devuelto por esta función de conversión será la dirección de una función que, cuando se invoque, tendrá el mismo efecto que la invocación del operador de llamada a función del tipo cierre.

Para ilustrar cómo una lambda puede admitir dicha conversión, consideremos el siguiente ejemplo. Define un objeto función `baz` que defina explícitamente el operador de conversión:

Ex2_15: Conversión a un puntero a función. Código en vivo @Wandboxline-numbers=on

```
#include <iostream>

void callWith10(void(* bar)(int)) { bar(10); }

int main() {
    struct {
        using f_ptr = void(*)(int);

        void operator()(int s) const { return call(s); }
        operator f_ptr() const { return &call; }

    private:
        static void call(int s) { std::cout << s << '\n'; };
    } baz;
```

[17] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#6

```
    callWith10(baz);
    callWith10([](int x) { std::cout << x << '\n'; });
}
```

En el programa anterior hay una función `callWith10` que toma un puntero a función. Luego lo llamamos con dos argumentos (líneas 18 y 19): el primero usa `baz`, que es un tipo objeto función que contiene el operador de conversión necesario; se convierte a `f_ptr`, que es el mismo que el parámetro de entrada para `callWith10`. Más tarde, tenemos una llamada con una lambda. En este caso, por debajo el compilador realiza las conversiones necesarias.

Dicha conversión puede ser útil cuando necesites llamar a una función estilo C que requiera alguna devolución de llamada. Por ejemplo, a continuación puedes encontrar código que llama a `qsort` de la biblioteca de C y usa una lambda para ordenar los elementos en orden inverso:

Ex2_16: Llamar a una función estilo C. Código en vivo @Wandbox

```
#include <cstdlib>
#include <iostream>

int main () {
    int values[] = { 8, 9, 2, 5, 1, 4, 7, 3, 6 };
    constexpr size_t numElements = sizeof(values)/sizeof(values[0]);

    std::qsort(values, numElements, sizeof(int),
         [](const void* a, const void* b) noexcept {
             return ( *(int*)b - *(int*)a );
         }
    );

    for (const auto& val : values)
        std::cout << val << ", ";
}
```

Como puedes ver en el ejemplo de código, se usa `std::qsort`, que solo toma punteros a función como comparadores. El compilador puede hacer una conversión implícita de la lambda sin estado que pasamos.

Un caso complicado

La sección está disponible solo en la versión completa del libro.

EFII - Expresión Funcional Inmediatamente Invocada

En la mayoría de los ejemplos que has visto hasta ahora, puedes observar que definí una lambda y luego la llamé.

Sin embargo, también puedes invocar una lambda inmediatamente:

Ex2_19: Llamar a una lambda inmediatamente. Código en vivo @Wandbox

```
#include <iostream>

int main() {
   int x = 1, y = 1;
   [&]() noexcept { ++x; ++y; }(); // <-- call ()
   std::cout << x << ", " << y;
}
```

Como puedes ver arriba, la lambda se crea y no se asigna a ningún objeto cierre. Pero luego se llama con `()`. Si ejecutas el programa, puedes esperar ver `2, 2` como resultado.

Este tipo de expresión puede resultar útil cuando se tiene una inicialización compleja de un objeto `const`.

```
const auto val = []() {
    /* varias líneas de código... */
}(); // ¡llámala!
```

Arriba, `val` es un valor constante de un tipo devuelto por una expresión lambda, es decir:

```
// val1 es int
const auto val1 = []() { return 10; }();

// val2 es std::string
const auto val2 = []() -> std::string { return "ABC"; }();
```

A continuación, puedes encontrar un ejemplo más largo en el que usamos EFII como una lambda auxiliar para crear un valor constante dentro de una función:

Ex2_20: EFII y generación de HTML. Código en vivo @Wandbox

```
#include <iostream>
#include <string>

void ValidateHTML(const std::string&) { }

std::string BuildAHref(const std::string& link, const std::string& text) {
    const std::string html = [&link, &text] {
        const auto& inText = text.empty() ? link : text;
        return "<a href=\"" + link + "\">" + inText + "</a>";
    }(); // call!

    ValidateHTML(html);

    return html;
}

int main() {
    try {
        const auto ahref = BuildAHref("www.leanpub.com", "Leanpub Store");
        std::cout << ahref;
    }
    catch (...) {
        std::cout << "bad format...";
    }
}
```

El ejemplo anterior contiene una función `BuildAHref` que toma dos parámetros y luego crea una etiqueta HTML `<a> </a>`. Basándonos en los parámetros de entrada, construimos

la variable `html`. Si el `texto` no está vacío, lo usamos como el valor HTML interno. De lo contrario, usamos el `link`. Queremos que la variable `html` sea `const`, pero es difícil escribir código compacto con las condiciones requeridas en los argumentos de entrada. Gracias a EFII, podemos escribir una lambda separada y luego marcar nuestra variable con `const`. Posteriormente, la variable se puede pasar a `ValidateHTML`.

Una nota sobre la legibilidad

La sección está disponible solo en la versión completa del libro.

Heredar de una lambda

Puede ser sorprendente verlo, pero también puedes derivar de una lambda.

Dado que el compilador expande una expresión lambda en un objeto función con `operator()`, entonces podemos heredar de este tipo.

Echa un vistazo al código básico:

Ex2_21: Heredar de solo una lambda. Código en vivo @Wandbox

```
#include <iostream>

template<typename Callable>
class ComplexFn : public Callable {
public:
    explicit ComplexFn(Callable f) : Callable(f) {}
};

template<typename Callable>
ComplexFn<Callable> MakeComplexFunctionObject(Callable&& cal) {
    return ComplexFn<Callable>(std::forward<Callable>(cal));
}

int main() {
    const auto func = MakeComplexFunctionObject([]() {
        std::cout << "Hello Complex Function Object!";
    });
    func();
}
```

En el ejemplo, existe la clase `ComplexFn` que se deriva de `Callable`, que es un parámetro de plantilla. Si queremos derivar de una lambda, necesitamos hacer un pequeño truco, ya que no podemos deletrear el tipo exacto del tipo cierre (a menos que lo envolvamos en `std::function`). Es por eso que necesitamos la función `MakeComplexFunctionObject` que puede realizar la deducción del argumento de plantilla y obtener el tipo cierre de la lambda.

El `ComplexFn`, aparte de su nombre, es simplemente un contenedor simple sin mucho uso. ¿Existen casos de uso para tales patrones de código?

Por ejemplo, podemos extender el código anterior y heredar de dos lambdas y crear un conjunto sobrecargado:

Ex2_22: Heredar de dos lambdas. Código en vivo @Wandbox

```
#include <iostream>

template<typename TCall, typename UCall>
class SimpleOverloaded : public TCall, UCall {
public:
    SimpleOverloaded(TCall tf, UCall uf) : TCall(tf), UCall(uf) {}

    using TCall::operator();
    using UCall::operator();
};

template<typename TCall, typename UCall>
SimpleOverloaded<TCall, UCall> MakeOverloaded(TCall&& tf, UCall&& uf) {
    return SimpleOverloaded<TCall, UCall>(std::forward<TCall> tf,
                                          std::forward<UCall> uf);
}

int main() {
    const auto func = MakeOverloaded(
        [](int) { std::cout << "Int!\n"; },
        [](float) { std::cout << "Float!\n"; }
    );
    func(10);
    func(10.0f);
}
```

Esta vez tenemos un poco más de código: derivamos de dos parámetros de plantilla, pero

también necesitamos exponer sus operadores de llamada a función explícitamente.

¿Por qué sucede eso? Es porque cuando se busca la función de sobrecarga correcta, el compilador requiere que los candidatos estén en el mismo ámbito.

Para entender eso, escribamos un tipo simple que se derive de dos clases base. El ejemplo también comenta dos declaraciones `using`:

Ex2_23: Error al derivar de dos clases. Código en vivo @Wandbox

```
#include <iostream>

struct BaseInt {
    void Func(int) { std::cout << "BaseInt...\n"; }
};

struct BaseDouble {
    void Func(double) { std::cout << "BaseDouble...\n"; }
};

struct Derived : public BaseInt, BaseDouble {
    //using BaseInt::Func;
    //using BaseDouble::Func;
};

int main() {
    Derived d;
    d.Func(10.0);
}
```

Tenemos dos clases base que implementan `Func`. Queremos llamar a ese método desde el objeto derivado.

GCC emite el siguiente error:

```
error: request for member 'Func' is ambiguous
```

Debido a que comentamos las instrucciones `using`, `::Func()` puede ser del ámbito de `BaseInt` o de `BaseDouble`. El compilador tiene dos ámbitos para buscar el mejor candidato y, según el estándar, no está permitido.

De acuerdo, volvamos a nuestro caso de uso principal:

`SimpleOverloaded` es una clase elemental y no está lista para producción. Echa un vistazo al capítulo de C++17 donde discutiremos una versión avanzada de este patrón. Gracias a varias características de C++17, podremos heredar de múltiples lambdas (gracias a las plantillas variádicas) y aprovechar una sintaxis más compacta.

Almacenar lambdas en un contenedor

La sección está disponible solo en la versión completa del libro.

Resumen

En este capítulo aprendiste a crear y usar expresiones lambda. Describí la sintaxis, la cláusula de captura, el tipo lambda y cubrimos muchos ejemplos y casos de uso. Incluso fuimos un poco más lejos y te mostré un patrón para derivar de una lambda o almacenarla en un contenedor.

¡Pero eso no es todo!

Las expresiones lambda se han convertido en una parte importante de C++ moderno. Con más casos de uso, los desarrolladores también vieron posibilidades de mejorar esta función. Y es por eso que ahora puedes pasar al siguiente capítulo y ver las actualizaciones esenciales que el comité internacional ISO para la normalización del lenguaje C++ agregó en C++14.

Referencias

Borradores del Estándar de C++

Aquí están los borradores finales de los estándares de C++, habitualmente con arreglos editoariales. Disponibles en timsong-cpp/cppwp[18].

Secciones sobre la expresión lambda:

- C++11 N3337 - [expr.prim.lambda][19]
- C++14 N4140 - [expr.prim.lambda][20]
- C++17 N4659 - [expr.prim.lambda][21]
- C++20 N4861 - [expr.prim.lambda][22]

Otras referencias

- Expresiones lambda - cppreference.com[23]
- Apoyo de compiladores de C++ - cppreference.com[24]
- Categorías de valor - cppreference.com[25]
- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 1ra edición por Scott Meyers, véase @Amazon.com[26]
- Functional Programming in C++: How to improve your C++ programs using functional techniques por Ivan Cukic, véase @Amazon[27]
- Microsoft Docs - Lambda Expressions in C++[28]

[18] https://github.com/timsong-cpp/cppwp
[19] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda
[20] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda
[21] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda
[22] https://timsong-cpp.github.io/cppwp/n4861/expr.prim.lambda
[23] https://es.cppreference.com/w/cpp/language/lambda
[24] https://es.cppreference.com/w/cpp/compiler_support
[25] https://es.cppreference.com/w/cpp/language/value_category
[26] https://amzn.to/2ZpA7yz
[27] https://amzn.to/3oEMVMY
[28] https://docs.microsoft.com/es-es/cpp/cpp/lambda-expressions-in-cpp

- Sticky Bits - Demystifying C++ lambdas[29]
- The View from Aristeia - Lambdas vs. Closures[30]
- Sy Brand - Passing overload sets to functions[31]
- Jason Turner - C++ Weekly - Ep 128 - C++20's Template Syntax For Lambdas[32]
- Jason Turner - C++ Weekly - Ep 41 - C++17's constexpr Lambda Support[33]
- Stack Overflow - c++ - Recursive lambda functions in C++11[34]
- Pedro Melendez - Recursive lambdas in C++(14)[35]
- Andreas Fertig - Under the covers of C++ lambdas - Part 2: Captures, captures, captures[36]
- Scott Meyers - Standard C++ - Universal References in C++11[37]
- Standard C++ Website - Quick Q: Why can noexcept generate faster code than throw()?[38]
- Bjarne Stroustrup - C++ Style and Technique FAQ[39]
- C++ Core Guidelines[40]
- Jonathan Boccara - How Lambdas Make Function Extraction Safer - Fluent C++[41]

[29] https://blog.feabhas.com/2014/03/demystifying-c-lambdas/
[30] http://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html
[31] https://blog.tartanllama.xyz/passing-overload-sets/
[32] https://www.youtube.com/watch?v=ixGiE4-1GA8&
[33] https://www.youtube.com/watch?v=kmza9U_niq4
[34] https://stackoverflow.com/questions/2067988/recursive-lambda-functions-in-c11
[35] http://pedromelendez.com/blog/2015/07/16/recursive-lambdas-in-c14/
[36] https://andreasfertig.blog/2020/11/under-the-covers-of-cpp-lambdas-part-2-captures-captures-captures/
[37] https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
[38] https://isocpp.org/blog/2014/09/noexcept-optimization
[39] https://stroustrup.com/bs_faq2.html
[40] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[41] https://www.fluentcpp.com/2020/11/13/how-lambdas-make-function-extraction-safer/