Programación de estructuras lineales

Tema 5. Excepciones, destructores y técnica RAII

Universidad Europea

Tema 5. Excepciones, destructores y técnica RAII

Índice

Tema 5. Excepciones, destructores y técnica RAII

1. Presentación

Cuando se crea un objeto de una clase, el **constructor** invocado debe establecer los invariantes de la misma —es decir, el conjunto de propiedades que deben respetarse independientemente del estado del objeto—, posiblemente adquiriendo **recursos** tales como memoria dinámica, archivos, *locks* (programación concurrente), *sockets* (*networking*), etc.

Una de las características definitorias del lenguaje C++ descansa en el hecho de que, al finalizar el tiempo de vida de un objeto, el destructor de la clase al que pertenece es implícitamente invocado de manera automática. La denominada **técnica RAII** (*Resource Acquisition is Initialization*), a la que dedicaremos nuestra atención en este tema, fue inventada por Bjarne Stroustrup como mecanismo de adquisición y liberación de recursos en C++ de forma segura ante excepciones. Intrínsecamente ligada a la mencionada propiedad de los destructores, esta técnica dicta que sea el destructor de la clase el encargado de liberar los recursos que el objeto haya adquirido durante su tiempo de vida y que aún se encuentren activos.

Como veremos en este tema, dicha liberación se lleva a cabo de manera determinista, de forma que los recursos no son utilizados más tiempo del estrictamente necesario, algo clave en sistemas de tiempo real. Por lo que respecta a los recursos de memoria, esto contrasta claramente con los mecanismos de recolección de basura disponibles en otros lenguajes de programación. La recolección de basura (en sus modalidades *mark-sweep* o *mark-compact*) se ha demostrado ciertamente innecesaria para la mayor parte de los programas (bien) escritos en C++, si bien se ha argumentado que su eventual inclusión en el lenguaje facilitaría en gran medida la implementación de estructuras de datos altamente concurrentes.

2. Destructores

A diferencia del constructor o constructores de una clase, el **destructor** es único. Esta función miembro especial se denota con el mismo nombre que la clase a la que pertenece, precedido por una tilde (~), y no puede poseer argumentos ni retornar valores. De no declararse explícitamente, el compilador generará un destructor por defecto (como miembro **inline public**) para la clase.

Cuando el destructor de una clase es invocado para un objeto, se ejecuta primeramente el cuerpo de la función. Cuando el flujo del programa alcanza el final de la misma (es decir, su llave de cierre }), todos los subobjetos (si los hubiese) son destruidos automáticamente en orden inverso al de su construcción a través de sus respectivos destructores.

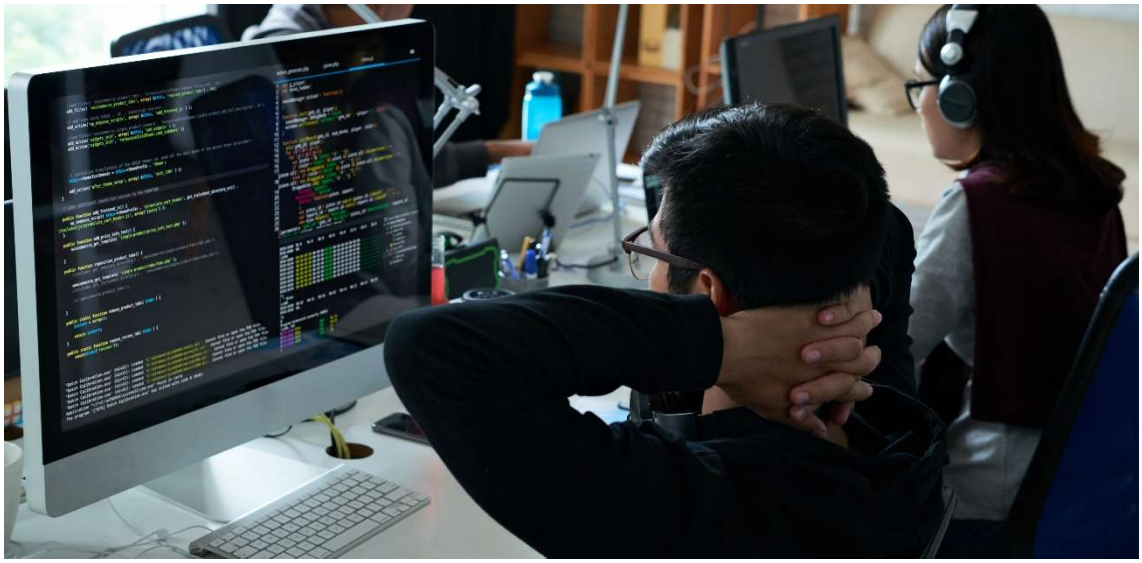
```
class X : /* clases base de X */ {
   // datos miembro
public:
   // ...
   ~X()
   {
      // cuerpo del destructor
      // ...
   } /* Punto de inicio de la secuencia de destrucción (ISO
C++11 12.4.8):
      (a) destruir datos miembro no-estáticos de X;
      (b) destruir las clases bases directas de X;
      Clases base y datos son destruidos en orden inverso a su
construcción */
};
```

A modo de ejemplo de la función destructor:

```
class Y {
public:
   ~Y();
   // ...
};

class BaseOne {
public:
   virtual ~BaseOne();
   // ...
};

class BaseTwo {
```

```
public:
    virtual ~BaseTwo();
    // ...
};

class X : public BaseOne, public BaseTwo {
   Y y1_,
     y2_;
public:
   X(Y const& y1, Y const& y2) : BaseOne{}, BaseTwo{}, y1_{y1},
y2_{y2} { }
   // ...
   ~X()
   {
      // cuerpo del destructor
   } // aquí se ejecuta la secuencia de destrucción:
     // y2_.~Y(), y1_.~Y(), BaseTwo::~BaseTwo(),
BaseOne::~BaseOne()
};
```

- La clase **X** deriva públicamente de las clases **BaseOne** y **BaseTwo** y posee dos atributos privados de tipo Y.
- El constructor de la clase **X** inicializa los subobjetos de las clases base y los atributos privados siguiendo la secuencia **BaseOne{}, BaseTwo{}, y1_{y1}, y2_{y2}.**
- Al finalizar la ejecución del cuerpo del destructor **~X(),** dichos subobjetos son destruidos automáticamente en orden inverso a su creación, según la secuencia **y2_.~Y(), y1_.~Y(), BaseTwo::~BaseTwo(), BaseOne::~BaseOne().**



3. Duración de almacenamiento

La siguiente tabla indica, a modo de recordatorio, el tiempo de vida de una variable en virtud de su tipo de **duración de almacenamiento:**

Tabla 1. Duraciones de almacenamiento.

Duración de almacenamiento	Tipo de variable afectada	Finalización del tiempo de vida
Estática	(i) Declarada en un ámbito *namespace* (incluido el espacio de nombres global). (ii) Declarada *static* o *extern*.	Al terminar la ejecución del programa.
Automática	Local, excepto si se declara *static*, *extern* o *thread_local*.	Cuando el flujo del programa abandona el ámbito donde la variable fue creada. En ese momento, los objetos con almacenamiento automático (variables locales) creados en dicho ámbito son destruidos en orden inverso a su construcción.
Dinámica	Alojada dinámicamente durante la ejecución del programa mediante una expresión de tipo *new*.	Al alcanzarse una expresión de tipo *delete*.
Thread-local	Declarada thead_local.	Al finalizar la ejecución del hilo donde la variable fue creada.

3.1. Ejemplo 1

A modo de ejemplo, consideremos el siguiente código puramente ilustrativo:

```
#include <iostream>

struct X {
   int i;
   X(int ii)
      : i{ii} { std::cout << "Construct X object #" << i <<
std::endl; }
   ~X() { std::cout << "Destruct X object #" << i << std::endl;
}
};

int main()
{
   X a{1};           // (A)
   X b{2};           // (B)
   X* p = new X{3}; // (C)
   static X c{4};    // (D)
   // ...
   delete p;         // (E)
   std::cout << "Exiting main function\n";
}                    // (F)

/* Output:
   "Construct X object #1" from (A)
   "Construct X object #2" from (B)
   "Construct X object #3" from (C)
   "Construct X object #4" from (D)
   "Destruct X object #3" from (E)
   "Exiting main function"
   "Destruct X object #2" and...
   "Destruct X object #1" from (F)
   "Destruct X object #4" at the end of program execution
*/
```

Las variables locales **a** y **b** (de tipo **X**) y p (de tipo X*) son inicializadas en (A), (B) y (C), respectivamente, poseyendo todas ellas una duración de almacenamiento automática. El objeto referenciado por el puntero **p** posee una duración de almacenamiento dinámica, siendo destruido y su memoria liberada en (E) mediante una expresión de tipo *delete*. El almacenamiento del puntero **p** aún prosigue tras (E), sin embargo, pudiendo ser reasignado con la dirección en memoria de otro objeto del tipo apropiado. El tiempo de vida de **a, b** y **p** acaba en (F), al salir estos fuera del ámbito donde fueron creados (en este caso, la función principal). Es en dicho punto cuando los objetos a y b son destruidos en orden inverso a su creación invocándose automáticamente al destructor de su clase. Respecto al objeto **c** de tipo **X** creado en

(D), este posee duración de almacenamiento estática, siendo destruido al finalizar la ejecución del programa.

Como regla general, y salvo en situaciones muy específicas, nunca debe llamarse explícitamente al destructor de una variable local. Para controlar su tiempo de vida, basta encerrarla en un ámbito adecuado (definido por un par cerrado de llaves **{ }**):

```
{
   X x1{1};
   {
      X x2{2};
      // ...
   }           // x2 es destruido aquí
   // ...
}              // x1 es destruido aquí
```

4. Excepciones

La **emisión de excepciones** es el mecanismo natural de información de los errores (como su nombre indica, *realmente excepcionales*) acontecidos durante la ejecución de un programa.

Una función que, por la razón que fuera, sea incapaz de completar satisfactoriamente su tarea debe informar del error emitiendo una excepción (típicamente un objeto de una clase definida a tal efecto) a través de una expresión de tipo **throw**.

El código que realice una llamada a dicha función deberá encerrase en un bloque **try** y capturar y manejar las excepciones emitidas por la misma a través de una o varias cláusulas **catch.**

El esquema básico de actuación es, pues, el siguiente:

```
struct Error { };

void g()
{
   // ...
   if (/* se cumple alguna condición de error */)
      throw Error{};
   // ...
}

void f()
{
   try {
      g();
   }
   catch (Error& e) {
      // aquí se maneja la excepción de tipo Error
```

```
    }
}
```

Se recomienda que las excepciones sean siempre capturadas por referencia.

Al emitirse una excepción dentro de un bloque **try-catch,** el control del flujo se transfiere desde el punto de lanzamiento de la misma hasta la primera cláusula **catch** que pueda manejarla. Al alcanzarse dicha cláusula, todos los objetos con almacenamiento automático que hayan sido creados desde el inicio del bloque **try** son destruidos en orden inverso a su creación (invocándose a los destructores de sus clases de forma automática), en un proceso de **desenredo de la pila.**

El destructor de una clase no debe emitir excepciones bajo ninguna circunstancia. Si un destructor llamado durante un desenredo de la pila lanza una excepción, la función **std::terminate** es invocada en tiempo de ejecución, lo que por defecto causa una interrupción anómala del programa.

La función **std::terminate** es también invocada de no encontrarse una cláusula *catch* capaz de manejar una excepción, en cuyo caso el que se produzca o no un desenredo de la pila es algo, según el estándar del lenguaje, dependiente de la implementación.

Si la excepción se produjese durante la construcción de un objeto que consta de subobjetos, se llamará a los destructores de los subobjetos que hayan sido creados satisfactoriamente antes de la emisión de la excepción, también en orden inverso a su creación.

4.1 Ejemplo 2

De acuerdo con lo dicho, la ejecución del siguiente programa produce la lista de mensajes indicada más abajo:

```
#include<iostream>

struct Error { };

struct X {
   X() { std::cout << "X's constructor" << std::endl; }
   ~X() { std::cout << "X's destructor" << std::endl; }
};

struct Y {
   Y() { std::cout << "Y's constructor" << std::endl; }
   ~Y() { std::cout << "Y's destructor" << std::endl; }
};

struct Z {
   Y y; // subobjecto
   Z() : y{} // inicializamos subobjecto 'y' (D)
   {
      std::cout << "Z's constructor "
                << "throws an Error type exception" <<
std::endl;
      throw Error{}; // (E)
   }
   ~Z() { std::cout << "Z's destructor" << std::endl; }
};

void f()
{
   std::cout << "f() function call" << std::endl;
   Z z; // (C) lanza excepción
   std::cout << "Exiting f() function" << std::endl; // esta
línea no llega a ejecutarse
}

int main()
{
   try {
      X x;      // (A)
      f();      // (B)
   }
   catch (Error const&) {  // (F)
      std::cerr << "Exception caught";
   }
}
```

```
/* Output:
   X's constructor
   f() function call
   Y's constructor
   Z's constructor throws exception of type Error
   Y's destructor
   X's destructor
   Exception caught */
```

- En su ejecución, el programa crea primeramente el objeto **x** de tipo **X** (véase el punto **A**).
- Se llama entonces a la función **f()** (en **B**), en la cual se invoca al constructor de la clase **Z** con el fin de inicializar el objeto **z** (en **C**).
- Dentro del constructor, el subobjeto **y** de tipo **Y** llega a inicializarse correctamente antes de producirse la excepción (véanse **D** y **E**).

Al emitirse esta, el control es transferido a la cláusula **catch** (en **F**) capacitada para manejar excepciones de tipo **Error,** proceso durante el cual el subobjeto **y** y el objeto **x** son destruidos (por ese orden) en el desenredo de la pila.

Observemos que el destructor de la clase **Z** no es invocado en ningún momento, puesto que el **objeto z nunca** llega **a existir.** En efecto, el tiempo de vida de un objeto empieza al alcanzarse con éxito la llave de cierre (}) del constructor que lo crea. Dado que la excepción es emitida en nuestro caso por el constructor de la clase Z durante la inicialización del objeto z, su creación queda inconclusa.

5. Garantías ante excepciones

Al implementar nuestros propios algoritmos y estructuras de datos en C++, deberemos analizar cuidadosamente el tipo de garantía que ofreceremos a los usuarios ante una eventual emisión de excepciones por parte de una operación dada. De acuerdo con la clasificación realizada por David Abrahams, uno de los desarrolladores principales de la familia de bibliotecas Boost, pueden distinguirse tres tipos fundamentales de garantías, a modo de acuerdo contractual entre las componentes y el usuario:

Tabla 2. Garantías.

Garantía básica	Se respetan los invariantes básicos de todos los objetos y se evita la fuga de recursos (memoria, ficheros, etc.). Ello permite el uso de los objetos (incluyendo su destrucción) con posterioridad a la emisión de la excepción.
Garantía fuerte	La operación concluye satisfactoriamente o emite una excepción, en cuyo caso el estado del programa retorna al inmediatamente anterior a la ejecución de la operación.
Garantía nothrow	La operación no emite excepciones bajo ninguna circunstancia.

Los algoritmos de la biblioteca estándar del lenguaje proporcionan, al menos, la garantía básica ante excepciones. Ciertos métodos como **std::vector<>::push_back(), std::uninitialized_copy(), std::uninitialized_fill()** o **std::**uninitialized**_fill_n()** proporcionan la garantía fuerte, mientras que operaciones como **std::swap()** no emiten excepciones bajo ninguna circunstancia.

6. Técnica RAII

Consideremos la función siguiente:

```
void arrayTest(std::size_t n)
{
   auto const p = new double[n];                    // (A)

   // utilizamos la matriz (este bloque de código puede emitir
excepciones)
   // ...

   delete[] p;                                      // (B)
}
```

En ella, se asigna dinámicamente memoria para una matriz unidimensional de escalares tipo **double** mediante una expresión de tipo **new.** Una vez completado un conjunto de operaciones sobre la matriz, el bloque de memoria es liberado mediante una expresión **delete. Este código no es seguro ante** excepciones. En efecto, observemos que de lanzarse una excepción entre la asignación de memoria en (**A**) y su posterior liberación en (**B**), el proceso de desenredo de la pila no liberaría la memoria referenciada por el puntero, pues la operación **delete[] p** no llegaría a ejecutarse. Con el fin de evitar esta posible laguna de memoria, analizaremos dos opciones; siendo la segunda la solución más idiomática:

1. **Solución 1 - Introducir un bloque de limpieza try-catch:**

El código intermedio que puede lanzar la excepción se encierra en un bloque **try** y la excepción se atrapa y se maneja convenientemente en un bloque **catch.** Aunque válida, esta solución complica innecesariamente el código (incluso en una situación tan sencilla como la considerada) y es poco eficiente.

```
void arrayTest(std::size_t n)
{
   auto const p = new double[n]; // (A)
   try {
      // utilizamos la matriz; este bloque de código puede
emitir excepciones...
   }
   catch (...) {  // capturamos cualquier posible excepción
emitida,
      delete[] p; // liberamos la memoria
      throw;      // y relanzamos la excepción fuera de la
función
```

```
    }
    delete[] p; // si no ocurrió ninguna excepción, liberamos la
memoria
}
```

Si el operador **new[]** no encontrara memoria suficiente para asignar en (**A**), se emitiría una excepción de tipo **std::bad_alloc.** Ante tal eventualidad, permitimos sencillamente que la excepción se propague fuera de la función sin necesidad de realizar operaciones adicionales (no existiría fuga de memoria, pues no habría terminado de adquirirse)

2. **Solución 2 - Utilizar la técnica RAII (Resource Acquisition Is Initialization)**

Diseñada por Bjarne Stroustrup, dicta que todo recurso debe quedar representado por un objeto local cuyo destructor se encargue de su liberación. En nuestro ejemplo, definimos una plantilla de clase **Dinarray<>,** capaz de encapsular matrices unidimensionales de tamaño conocido en tiempo de ejecución. El constructor de la clase adquiere memoria dinámica para la matriz mediante una expresión **new** y su destructor la libera mediante una expresión **delete.**

```
template<typename T>
class Dynarray {
    T* p_;              // puntero al primer elemento de la matriz
    std::size_t size_; // número de elementos en la matriz
public:
    // construcción y destrucción:
    explicit Dynarray(std::size_t n) : p_{new T[n]}, size_{n} { }
    ~Dynarray() { delete[] p_; }

    // acceso a elementos (sin control de acceso):
    T const& operator[](std::size_t i) const { return p_[i]; }
    T&       operator[](std::size_t i)       { return p_[i]; }

    // resto de la interfaz pública, incluyendo operaciones de
copia, iteradores, etc
};
```

Haciendo uso de esta plantilla, la función **arrayTest()** se reduce simplemente a:

```
void arrayTest(std::size_t n)
{
    Dynarray<double> d(n); // construimos un array de n elementos
    // utilizamos el array...

} // Dynarray<double>::~Dynarray() libera la memoria
automáticamente en este punto
```

De emitirse una excepción una vez creada la matriz, el destructor de la clase **Dynarray<double>** será invocado durante el desenredo de la pila, liberándose la memoria dinámica asignada por el constructor. Asimismo, de no producirse emisión de excepciones, la función **arrayTest()** terminará de ejecutarse normalmente y el destructor de la clase será llamado de forma automática al salir el objeto d fuera de ámbito, liberándose también la memoria como en el caso anterior. En cualquiera de los casos, pues, es el destructor el encargado de invocar, de manera automática y determinista, al operador **delete[].** Se trata de una solución natural y eficiente que caracteriza, más que ninguna otra, al lenguaje C++.

Finalmente, aunque a primera vista pudiera parecer que esta solución requiere un mayor esfuerzo de codificación, debemos tener en cuenta que, una vez implementada en un fichero de cabecera, la plantilla **Dynarray<>** puede ser reutilizada en múltiples proyectos. Por supuesto, se recomienda utilizar las estructuras de datos proporcionadas por el propio estándar del lenguaje, en este caso la plantilla **std::vector<>** disponible en el fichero de cabecera **<vector>.**

7. Resumen

En este tema hemos analizado el modo de funcionamiento del destructor de una clase, responsable de liberar de forma automática los recursos abiertos que aún se encuentren gestionados por un objeto al finalizar su duración de almacenamiento.

Asimismo, hemos explicado el modo de comunicar errores en la ejecución de funciones (por ejemplo en los constructores de una clase) mediante la emisión de excepciones.

La **técnica RAII,** en particular, es una solución idiomática en C++ que permite crear código seguro ante excepciones utilizando el modo de funcionamiento de los destructores. Esta técnica no se circunscribe únicamente a la asignación dinámica de memoria, sino que afecta a cualquier tipo de recurso, sea este un fichero, *mutex* (programación concurrente), *socket* (*networking*), etc. Este tratamiento homogéneo de recursos en C++ contrasta con el de otros lenguajes de programación orientados a objetos, en los que debe distinguirse claramente entre los mecanismos de recolección de basura (ligados a la gestión de la memoria dinámica) y los bloques **try-catch-finally** (a utilizar con otros tipos de recursos).

Referencias bibliográficas

Stroustrup B. (2013). *The C++ Programming Language* (cuarta edición). (Pág. 377). USA: Addison-Wesley.