Programación de estructuras lineales

Tema 7. Clase Dynarray<>. Semántica de copia

Universidad Europea

Tema 7. Clase Dynarray<>. Semántica de copia

Índice

Tema 7. Clase Dynarray<>. Semántica de copia

1. Presentación

En este capítulo analizaremos en detalle la implementación de una plantilla de clase **Dynarray<>** que represente matrices unidimensionales de objetos almacenadas en la memoria libre. Todos los objetos en el contenedor pertenecerán a un mismo tipo seleccionado por el programador en tiempo de compilación.

Esta primera estructura de datos nos permitirá discutir varios aspectos fundamentales que aparecerán de forma recurrente en futuras implementaciones, a saber:

- Explicar la distinción entre **inicialización** y **asignación**.
- Entender la **semántica de copia** en el lenguaje C++.
- Crear una clase cuyo uso resulte seguro ante el lanzamiento de excepciones.

2. Inicialización vs asignación

- **Inicialización.** Cuando implementamos una clase genérica **Complex<>** capaz de representar números complejos, omitimos deliberadamente los detalles relacionados con la **inicialización de objetos** a partir de otros ya existentes. En el código siguiente, por ejemplo, se crea un número complejo **y** (previamente no existente) como copia exacta de otro ya existente **x**:

```
Complex<double> x{1.0,2.0},   // x = (1,2)
    y{x};         // y = (1,2) también; equivalente a: auto y =
x;
```

- **Asignación.** Una operación distinta consiste en la **asignación de objetos** en la forma siguiente:

```
Complex<double> x{1.0,2.0},   // x = (1,2)
                y{3.0,4.0};   // y = (3,4)
y = x;                        // y es ahora una copia de x, es
decir, x = y = (1,2)
```

La diferencia principal con respecto al primer código consiste en que, aquí, **y** es un objeto ya inicializado cuyo valor original es eliminado con el fin de almacenar una copia exacta de **x**.

Aun cuando no se hizo mención explícita de tales operaciones, estas son ciertamente posibles con cualquier instancia de la plantilla **Complex<>**. Ello se debe a que el compilador define implícitamente dos funciones miembro **inline public** especiales para la clase:

- Un **constructor copia**, que permite inicializar un objeto copiando la representación de otro dado (véase el primer bloque de código de ejemplo anterior):

```
template<typename T> inline
Complex<T>::Complex(Complex<T> const&);
```

- Un **operador de asignación copia**, que permite copiar la representación de un objeto dado en otro ya existente (véase el segundo bloque de ejemplo anterior):

```
template<typename T> inline
Complex<T>& Complex<T>::operator=(Complex<T> const&);
```

Observemos, en este último caso, como el valor retornado es una referencia al propio objeto con el fin de permitir la concatenación de asignaciones del tipo **x = y = z** (equivalente a **x.operator=(y.operator=(z))**).

Aunque la sintaxis pueda confundir al programador principiante en C++, **Complex<double> y = x**; (o equivalentemente, utilizando inferencia automática de tipos, **auto y = x;**) implica una **inicialización**, invocándose al constructor copia y no al operador de asignación. En ese sentido, la sentencia es equivalente a **Complex<double> y{x};**. Si bien resulta completamente innecesario en el caso de la plantilla Complex<>, podríamos definir explícitamente las funciones miembro anteriores como:

```
template<typename T>
class Complex {
   static_assert(std::is_floating_point<T>::value,
                 "Value type must be float, double, or long
double");
   T re_,
     im_;
public:
   // ...
```

```
    Complex(Complex<T> const& c) // constructor copia
        : re_{c.re_}, im_{c.im_} { }

    Complex<T>& operator=(Complex<T> const& c) // operador de
asignación copia
    {
        re_ = c.re_;
        im_ = c.im_;
        return *this; // permite la asignación en cadena
    }
};
```

Independientemente del parámetro utilizado (**float, double o long double**), la clase **Complex** es lo suficientemente sencilla (en particular, carece tanto de funciones miembro virtuales como de clases base y sus datos miembro privados son de tipo primitivo) como para que tanto el **constructor copia** como el **operador de asignación copia** generados por el compilador sean **triviales**, es decir, realicen la copia de un objeto *byte* a *byte*, de manera análoga a si se utilizase la función de bajo nivel **std::memmove().**

En este tema aprenderemos cuándo y cómo definir constructores copia y operadores de asignación copia para nuestras clases. Como ejercicio básico que nos permita analizar al detalle las peculiaridades de la **semántica de copia** en C++, implementaremos una plantilla de clase **Dynarray<>** que encapsule matrices unidimensionales de tamaño conocido en tiempo de ejecución. Por supuesto, prestaremos también especial atención a las garantías ofrecidas por sus funciones miembro ante la emisión de excepciones.

3. Estructura Dynarray<>

Nos proponemos implementar una plantilla de clase **Dynarray<>** que encapsule matrices dinámicas unidimensionales (de tamaño fijo) con elementos de tipo genérico **T:**

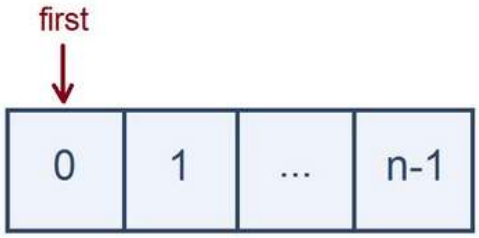

Figura 1. Array unidimensional de n elementos.

Tema 7. Clase Dynarray<>. Semántica de copia

Nuestro estudio tendrá una motivación puramente educativa, por supuesto, de forma que algunas de las elecciones en la implementación de la plantilla serán ciertamente discutibles y mejorables. Como es natural, como programadores deberíamos hacer uso de los contenedores ofrecidos por nuestros compiladores atendiendo a sus condiciones de uso y de distribución.

Siguiendo la técnica RAII, un constructor de la clase será el encargado de asignar la memoria dinámica, mientras que el destructor la liberará al concluir la duración de almacenamiento del objeto.

```
template<typename T> // el parámetro T debe ser un tipo
class Dynarray {
   std::size_t size_; // número de elementos en el array
   T* first_;         // puntero al primer elemento en el array
public:
   // definiciones de tipos:
   using value_type             = T;
   using const_reference        = T const&;
   using reference              = T&;
   using const_iterator         = T const*;
   using iterator               = T*;
   using const_reverse_iterator =
std::reverse_iterator<const_iterator>;
   using reverse_iterator       =
std::reverse_iterator<iterator>;
   using size_type              = std::size_t;

   // constructores:
   Dynarray(size_type size = 0);                    //
constructor #1
   Dynarray(size_type size, const_reference val);   //
constructor #2

   // destructor:
   ~Dynarray();

};
```

Observemos la introducción de múltiples **declaraciones *using*** en la interfaz pública de la clase. Así, por ejemplo, **value_type** es declarado como alias del tipo de elemento T almacenado en la matriz, **const_reference** es sinónimo de una referencia a un objeto constante de tipo T, etcétera. Nuestro objetivo al proceder así es doble. En primer lugar, proporcionamos un listado coherente de tipos, compatible con la biblioteca estándar del lenguaje, que el usuario puede utilizar para declarar variables en su propio código. En segundo lugar, mejoramos la sintaxis de la interfaz pública, la cual, como comprobaremos a continuación, se torna más precisa. A continuación se explican en más detalle la implementación de los constructores y del destructor de la clase.

4. Constructores y destructor

4.1. Primer constructor

```
template<typename T> inline
   Dynarray<T>::~Dynarray(size_type size = 0)
      : size_{size}, first_{(size_)? new T[size_] : nullptr}
   { }
```

En función del valor del entero sin signo **size** pasado como argumento, el constructor inicializa el puntero first (que apunta al primer elemento de la matriz) mediante una de las dos operaciones siguientes:

- De ser **size** mayor que cero, el constructor asigna un espacio en memoria suficiente para almacenar dicho número de elementos de tipo **T**. Esta operación puede provocar la emisión de una excepción **std::bad_alloc** de agotarse la memoria. Si **value_type** es un tipo primitivo, no se asignan valores a los elementos (estos quedan indeterminados), mientras que, si **value_type** es una clase, se invoca al constructor por defecto **value_type::value_type()** de la misma. Asumimos aquí, pues, que **value_type** es un tipo que puede construirse por defecto.

- Si **n** es cero, **first** es inicializado como puntero nulo. En este caso, el constructor no emitirá excepciones de ninguna clase.

4.2. Segundo constructor

```
template<typename T> inline
   Dynarray<T>::~Dynarray(size_type size, const_reference val)
      : Dynarray{size}
   {
      try {
         for (std::size_t i = 0; i < size_; ++i)
            first_[i] = val;
      }
      catch (...) {
         delete[] first_;
         throw;
      }
   }
```

Construye una matriz con **size** copias del valor **val** ofreciendo la garantía fuerte ante excepciones. En primer lugar, este segundo constructor delega en el primero la inicialización de una matriz de tamaño **size** para, a continuación, realizar las copias a través de un bucle **for**. Dicho bucle se encierra en un bloque **try** con el fin de que, si alguna de las asignaciones fallase, se capture la excepción y se destruyan las copias realizadas hasta ese momento, liberándose la memoria ocupada por la matriz.

Finalizaremos esta sección de la implementación de la matriz con la definición del **destructor** de la clase, encargado de destruir los elementos en la matriz y liberar la memoria asignada por el constructor de la clase base:

```
template<typename T> inline
Dynarray<T>::~Dynarray()
{
   delete[] first_;
}
```

5. Semántica de copia

Con la implementación del contenedor **Dynarray<>** del que disponemos a partir del estudio realizado en pantallas anteriores, cabe preguntarse si el **constructor copia** y el **operador de asignación copia** generados automáticamente por el compilador son adecuados para nuestra plantilla. A poco que reflexionemos sobre esta cuestión concluiremos que ambas funciones especiales generadas por omisión son en realidad inadmisibles en nuestro caso, al reducirse ambas a una mera copia *byte* a *byte* de la representación del objeto pasado como argumento, es decir, del entero **size_** y del puntero **first_** del objeto a copiar. Este comportamiento resulta funcionalmente equivalente a:

```
template<typename T> inline // constructor copia por defecto
Dynarray<T>:: Dynarray(Dynarray<T> const& d)
   : size_{d.size_}, first_{d.first_},
{ }

template<typename T> inline // operador de asignación copia por
defecto
Dynarray<T>& Dynarray::operator=(Dynarray<T> const& d)
{
   size_ = d.size_;
   first_ = d.first_;
   return *this;
}
```

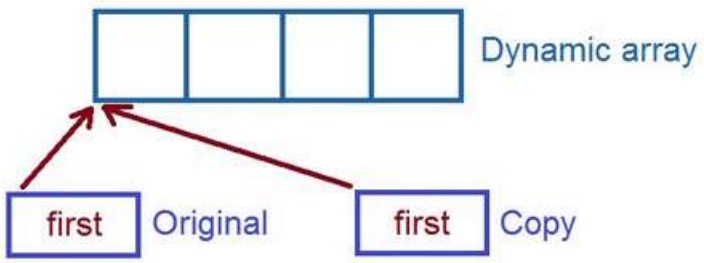

Figura 2. Punteros referenciando una misma localización en memoria.

Así, al inicializarse la copia de un objeto, tanto el original como la copia harían referencia al mismo bloque de memoria dinámica.

De ser destruido uno de estos dos objetos, por ejemplo, el restante quedaría invalidado, pues referenciaría un área de memoria previamente liberada por el destructor de la clase. Cualquier intento de acceder a los elementos de dicha matriz, o de destruir este segundo objeto, conllevaría efectos desastrosos en tiempo de ejecución. El caso del operador de asignación es si cabe aún más grave que el del constructor copia, pues la matriz inicialmente referenciada por el objeto de destino no es siquiera destruida antes de producirse la asignación, produciéndose una fuga de memoria. Es, pues, responsabilidad del programador el definir las operaciones correctas, analizadas en la siguiente pantalla.

El **constructor copia** debe asignar un espacio en memoria dinámica suficiente para contener tantos elementos como haya en la matriz origen. Se procederá entonces a realizar una copia de cada uno de dichos elementos en la nueva matriz de destino. La implementación es, en este caso, trivial:

```
template<typename T> inline
Dynarray<T>::Dynarray(Dynarray<T> const& d) // constructor copia
   : Dynarray{d.size_}
{
   try {
     for (std::size_t i = 0; i < size_; ++i)
        first_[i] = d.first_[i];
   }
   catch (...) {
      delete[] first_;
      throw;
   }
```

El **operador de asignación copia** producirá también un duplicado de la matriz pasada como argumento, si bien debe destruir la matriz inicialmente referenciada por el objeto de destino. Es deseable proporcionar la garantía fuerte ante excepciones, de modo que, de producirse una excepción durante el proceso de copia, se mantenga la representación inicial del objeto. Este comportamiento puede lograrse mediante la denominada **técnica copy-and-swap** (copiar e intercambiar):

```
template<typename T> inline
Dynarray<T>& Dynarray<T>::operator=(Dynarray<T> const& d)
{
   Dynarray<T> tmp{d};
   std::swap(size_,tmp.size_);
   std::swap(first_,tmp.first_);
   return *this;
}
```

En primer lugar, construimos un nuevo objeto **Dynarray<T>** de nombre **tmp** en el cuerpo del operador de asignación copia, inicializado a partir del objeto **d** pasado como argumento. Para ello, invocamos al constructor copia definido anteriormente. A continuación, intercambiamos las representaciones del objeto de destino ***this** y el temporal **tmp** llamando a la función estándar **std::swap()** disponible a tal efecto.

Esta simplemente intercambia tanto los punteros **first_** como los enteros **size_** de ambos objetos. Al abandonar el operador de asignación, el objeto **tmp,** y con él la matriz originalmente referenciada por el objeto de destino, es destruido automáticamente. La operación de intercambio **swap()** es **noexcept**. De emitirse una excepción durante la construcción del temporal **tmp**, esta se propaga fuera del operador de asignación sin que se llegue a modificar la representación original del objeto de destino.

Observemos que hemos preferido no proteger el cuerpo del operador de asignación copia contra operaciones de auto-asignación dada su escasa probabilidad de ocurrencia y el coste de la secuencia de control **if (this != &d) { /* ... */ }**, innecesaria en todos los demás casos.

6. Resto de la interfaz pública

Mostraremos a continuación el código completo de la plantilla Dynarray<> (que debería incluirse tanto en un fichero de cabecera como en un espacio de nombres convenientes), introduciendo funciones miembro que permitan iterar a través de la matriz, acceder a sus elementos (con y sin verificación de rango), etcétera. Se indican, asimismo, todos los ficheros de cabecera estándar necesarios para una correcta compilación. La mayor parte del código es autoexplicativo y descansa en las discusiones realizadas en pantallas anteriores:

```
#include <cstddef>
#include <initializer_list>
#include <iterator>
#include <limits>
#include <memory>
#include <new>
#include <stdexcept>
#include <type_traits>
#include <utility>

template<typename T>
class Dynarray {
   std::size_t size_;
   T* first_;

   void rangeCheck(size_type i) const
   {
      if (i >= size)
         throw std::out_of_range{"Out of range index"};
   }
public:
   // typedefs:
   using value_type             = T;
   using const_reference        = T const&;
   using reference              = T&;
```

```
   using const_iterator         = T const*;
   using iterator               = T*;
   using const_reverse_iterator =
std::reverse_iterator<const_iterator>;
   using reverse_iterator       =
std::reverse_iterator<iterator>;
   using size_type              = std::size_t;

   // construcción/copia/destrucción:
   Dynarray(size_type size = 0)
      : size_{size}, first_{(size_)? new T[size_] : nullptr}
   { }

   Dynarray(size_type size, const_reference val)
      : Dynarray{size}
   {
      try {
         for (std::size_t i = 0; i < size_; ++i)
            first_[i] = val;
      }
      catch (...) {
         delete[] first_;
         throw;
      }
   }

   Dynarray(Dynarray<T> const& d)
      : Dynarray{d.size()}
   {
      try {
         for (std::size_t i = 0; i < size_; ++i)
            first_[i] = d.first_[i];
      }
      catch (...) {
         delete[] first_;
         throw;
      }
   }

   Dynarray<T>& operator=(Dynarray<T> const& d)
   {
      // copy-and-swap idiom (strong exception safety
guarantee):
      Dynarray<T> tmp{d};
      std::swap(size_,tmp.size_);
      std::swap(first_,tmp.first_);
      return *this;
   }

   ~Dynarray() { delete[] first_; } // destructor

   // iteradores:
   const_iterator         begin()   const { return first_; }
   iterator               begin()         { return first_; }
```

```
   const_iterator         end()     const { return first_ +
size_; }
   iterator               end()           { return first_ +
size_; }
   const_iterator         cbegin()  const { return first_; }
   const_iterator         cend()    const { return first_ +
size_; }
   const_reverse_iterator rbegin()  const
                                    { return
const_reverse_iterator{cend()}; }
   reverse_iterator       rbegin()        { return
reverse_iterator{end()}; }
   const_reverse_iterator rend()    const
                                    { return
const_reverse_iterator{first_}; }
   reverse_iterator       rend()          { return
reverse_iterator{first_}; }
   const_reverse_iterator crbegin() const
                                    { return
const_reverse_iterator{cend()}; }
   const_reverse_iterator crend()   const
                                    { return
const_reverse_iterator{first_}; }

   // capacidad:
   size_type size()  const noexcept { return size_; }
   bool      empty() const noexcept { return size_ == 0; }
   // acceso a elementos:
   const_reference operator[](size_type i) const { return
first_[i]; }
   reference       operator[](size_type i)       { return
first_[i]; }

   const_reference at(size_type i) const { rangeCheck(i); return
first_[i]; }
   reference       at(size_type i)       { rangeCheck(i); return
first_[i]; }

   // modificadores:
   void swap(Dynarray& d) noexcept
   {
      std::swap(size_,d.size_);
      std::swap(first_,d.first_);
   }
};
```

Muy en particular, las sobrecargas del operador de indexación **operator[]** permiten acceder a los elementos de la matriz utilizando la sintaxis habitual del lenguaje C.

Así, por ejemplo, la operación **std::cout << m[5];** imprimiría el sexto elemento de la matriz **m**, siendo enteramente equivalente a una llamada al operador de indexación **std::cout << m.operator[](5);**. Las funciones miembro **at()** proporcionan acceso con control de acceso: de sobrepasar el índice el tamaño de la matriz, se emitirá una excepción de tipo **std::out_of_range**.

7. Funciones begin() y end(). Bucles for basados en rango

Figura 3. Funciones begin() and end().

Las funciones miembro públicas **begin()** y **end()** permiten iterar a través de la matriz unidimensional en el sentido habitual, desde el primer elemento al último, es decir, según la secuencia de índices de acceso **0,1,2,…,n-1,** siendo **n** el número total de entradas en la matriz.

La función **begin()** retorna un iterador que apunta al primer elemento de la matriz, mientras que **end()** lo hace a un hipotético elemento de tipo value_type posterior a la última entrada de la matriz (véase la figura superior). Las versiones constantes de dichas funciones, invocables para matrices constantes, retornan punteros a elementos constantes, de modo que la matriz no pueda modificarse.

Así operan también, por definición, **cbegin()** y **cend()**, si bien estas funciones pueden ser invocadas para objetos no constantes. Las funciones **rbegin()** y **rend()** (así como **crbegin()** y **crend())** permiten iterar la matriz en orden inverso (desde el último elemento al primero, según la secuencia de acceso n-1,n-2,…,2,1,0). El siguiente bloque de código proporciona varios ejemplos de uso:

```
Dynarray<int> d{5};
d[0] = 0;
d[1] = 1;
d[2] = 2;
d[3] = 3;
d[4] = 4;
for (auto p = d.cbegin(); p != d.cend(); ++p)
   std::cout << *p << ' ';  // output: 0 1 2 3 4
std::cout << std::endl;
for (auto p = d.begin(); p != d.end(); ++p)
   *p *= 2;                 // multiplicamos cada entrada por 2
for (auto p = d.crbegin(); p != d.crend(); ++p)
   std::cout << *p <<' ';   // output: 8 6 4 2 0
```

Conviene resaltar aquí la sintaxis simplificada que el nuevo estándar del lenguaje ofrece para recorrer por iteración cualquier rango definido por un par de funciones

begin() y **end()**. Nos referimos a la denominada **sentencia de control for basada en rango**. Así, los dos primeros bucles del código anterior podrían reescribirse como:

```
Dynarray<int> d{0,1,2,3,4};

for (auto x : d)
   std::cout << x << ' '; // output: 0 1 2 3 4

std::cout << std::endl;

for (auto& x : d)
   x *= 2;                 // multiplicamos cada entrada por 2
```

- La sentencia **for (auto x : d)** en el primer bucle de ejemplo puede leerse como "**para cada elemento x en el contenedor d**". En este caso, se realiza una copia de cada elemento (se habla entonces de **acceso por valor**) para enviarla a la salida estándar.
- Si la copia de las entradas de la matriz fuese una operación costosa y, como en el caso considerado, no fuese necesario modificar sus valores, convendría tomar referencias a elementos constantes, escribiéndose en su lugar **for (auto const& x : d)** (**acceso por referencia constante**). En nuestro caso no es necesario proceder así, dado que trabajamos con enteros **int**: ¡una referencia puede ser más costosa que una copia!
- En el segundo bucle de ejemplo, se toma una referencia no-constante de cada elemento de la matriz con el fin de poder modificar su valor, optándose por la forma for **(auto& x : d)** (**acceso por referencia**).

8. Resumen

En este Tema hemos analizado la implementación completa de la estructura de datos **Dynarray<>**. Aún siendo extremadamente sencilla, emplearemos el diseño de esta clase como modelo base para futuras estructuras de datos, muy en particular el vector de datos de longitud variable en tiempo de ejecución (consúltese la funcionalidad de la plantilla **std::vector<>** en el estándar de C++ para más detalles).

Te recomendamos familiarizarte con el uso combinado de iteradores y bucles basados en rango mediante la resolución de ejercicios prácticos y la codificación de programas simples como el proporcionado en el primer ejemplo propuesto en este tema. Por supuesto, revisaremos tales conceptos en siguientes unidades del curso para afianzar conocimientos.

Referencias bibliográficas

Stroustrup, B. (2013).*The C++ Programming Language*. USA: Addison-Wesley.

Enlaces de interés

Copy-and-swap idiom. <stackoverflow.com>

Range-for loop. <cppreference.com>