Comparación entre los *arrays* de la biblioteca estándar y los *arrays* predefinidos en C++

Vicente Benjumea, Manuel Roldán

Resumen

El estudio de *arrays* es parte importante de cualquier primer curso inicial de programación. Tradicionalmente este aspecto ha sido tratado (tanto en *C* como en *C++*) mediante *arrays* predefinidos. Sin embargo, desde el estándard C++11 (previamente en *TR1*), *C++* dispone de un tipo `array` proporcionado por la biblioteca estándar que aporta numerosas ventajas docentes sobre el uso tradicional de *arrays* predefinidos. Este documento presenta las características de cada uno y analiza las ventajas e inconvenientes de cada enfoque.

Arrays

En general, el tipo *array* se utiliza para definir una secuencia de un número determinado (definido en tiempo de compilación) de elementos de un mismo tipo de datos (simple o compuesto), de tal forma que se puede acceder a cada elemento individual de la secuencia de forma *parametrizada* mediante índices (el valor cero indica la posición del primer elemento de la colección).

Tanto en C como en C++ se pueden utilizar *arrays* predefinidos, pero en C++, desde la versión del estándar C++11 (previamente en *TR1*) es posible, además, usar el tipo `array` proporcionado por la biblioteca estándar de C++.

A continuación se muestran las características proporcionadas por ambos tipos de *arrays*.

Inicialización de variables y constantes

Ambos tipos de *arrays* permiten inicializar tanto una variable como una constante con una lista de valores.

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
const ABStd DATOS = {{ 1, 2, 3, 4, 5 }} ;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
const APredef DATOS = { 1, 2, 3, 4, 5 } ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
}
```

En el caso de *arrays* de la biblioteca, también se pueden inicializar tanto variables como constantes con los valores de otras variables o constantes. Sin embargo, en el caso de los *arrays* predefinidos no es posible (no es consistente con el resto de tipos de C++).

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
const ABStd DATOS = {{ 1, 2, 3, 4, 5 }} ;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    ABStd a2 = a1 ;     // Correcto
    ABStd a3 = DATOS ;  // Correcto
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
const APredef DATOS = { 1, 2, 3, 4, 5 } ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    APredef a2 = a1 ;       // ErrorCompilación
    APredef a3 = DATOS ;    // ErrorCompilación
}
```

Adicionalmente, en el caso de los *arrays* predefinidos, también se pueden definir variables y constantes sin especificar su tipo, y donde el número de elementos se deduce de los valores de la lista de inicialización. No obstante, esta característica no es deseable durante el aprendizaje de la programación, ya que no se proporciona una forma fácil para calcular el número de elementos del *array*.

```
    // No disponible en arrays de la biblioteca
```

```
const int DATOS[] = { 1, 2, 3, 4, 5 } ;
int main()
{
    int a1[] = { 1, 2, 3, 4, 5 } ;
}
```

Asignación de valores a variables

En el caso de *arrays* de la biblioteca, sí es posible asignar a una variable tanto una lista de valores, como el valor de otras variables o constantes. Sin embargo, en el caso de los *arrays* predefinidos no es posible (no es consistente con el resto de tipos de C++).

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
const ABStd DATOS = {{ 1, 2, 3, 4, 5 }} ;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    ABStd a2 ;
    a2 = {{ 1, 2, 3, 4, 5 }} ;  // Correcto
    a2 = a1 ;                   // Correcto
    a2 = DATOS ;                // Correcto
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
const APredef DATOS = { 1, 2, 3, 4, 5 } ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    APredef a2 ;
    a2 =  { 1, 2, 3, 4, 5 } ;   // ErrorCompilación
    a2 = a1 ;                   // ErrorCompilación
    a2 = DATOS ;                // ErrorCompilación
}
```

Operadores relacionales

En el caso de *arrays* de la biblioteca, si el tipo del elemento es comparable, entonces también se pueden comparar variables y constantes entre sí, tanto comparaciones de igualdad como comparaciones de relación (`==`, `!=`, `>`, `>=`, `<`, `<=`). También es posible compararlos con una lista de valores.

Sin embargo, en el caso de los *arrays* predefinidos, la comparación con otras variables y constantes compila correctamente, pero no produce los resultados esperados, ya que no compara los elementos, sino que compara las direcciones de las variables y constantes. Este comportamiento no es consistente con las comparaciones de otros tipo de datos, y por lo tanto, es propenso a errores. Además, no es posible comparar con una lista de valores.

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
const ABStd DATOS = {{ 1, 2, 3, 4, 5 }} ;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    ABStd a2 = {{ 1, 2, 3, 4, 5 }} ;
    if (a1 == a2) {          // Correcto
        cout << "Iguales" ;  // Muestra Iguales
    } else {
        cout << "Distintos" ;
    }
    if (a1 > DATOS) {        // Correcto
        cout << "Mayor" ;    // Muestra MenorIgual
    } else {
        cout << "MenorIgual" ;
    }
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
const APredef DATOS = { 1, 2, 3, 4, 5 } ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    APredef a2 = { 1, 2, 3, 4, 5 } ;
    if (a1 == a2) {          // ErrorEjecución
        cout << "Iguales" ; // muestra Distintos
    } else {
        cout << "Distintos" ;
    }
    if (a1 > DATOS) {        // ErrorEjecución
        cout << "Mayor" ;   // Muestra Mayor
    } else {
        cout << "MenorIgual" ;
    }
}
```

Cálculo del número de elementos del array

Tanto en los *arrays* predefinidos, como en los *arrays* de la biblioteca estándar, sí es posible conocer el número de elementos de un *array* a través del valor de la constante que usualmente se utiliza para la

defínción de su tipo.

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    for (int i = 0; i < NELMS; ++i) {
        cout << a1[i] << endl;
    }
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    for (int i = 0; i < NELMS; ++i) {
        cout << a1[i] << endl;
    }
}
```

Adicionalmente, los *arrays* de la biblioteca estándar también proporcionan un método (`size`) para conocer el número de elementos de una variable o constante. Este mecanismo es muy adecuado, ya que es independiente de como haya sido declarada la variable, o de como haya sido definido el tipo, por lo que reduce dependencias y hace que el código sea más seguro y menos propenso a errores.

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    for (unsigned i = 0; i < a1.size(); ++i) {
        cout << a1[i] << endl;
    }
}
```

Acceso a los elementos

Cuando se accede a un elemento de un *array*, mediante el acceso por posición, tanto en los *arrays* predefinidos como en los *arrays* de la biblioteca estándar, el código C++ no comprueba si la posición se encuentra dentro del rango correcto (desde cero hasta el número de elementos menos uno). Esto es así por cuestiones de eficiencia en estas operaciones básicas.

Sin embargo, la ausencia de comprobación del acceso fuera de rango causa numerosos errores durante la ejecución del código, y hace el código menos seguro.

No obstante, los *arrays* de la biblioteca estándar proporcionan un *modo de depuración* (que se activa en el caso del compilador `g++` con la opcion `-D_GLIBCXX_DEBUG`) que proporciona un control de acceso fuera de rango cuando se accede a los elementos de un *array* de la biblioteca estándar. Este control de acceso *aborta* la ejecución del programa (lanza una *excepción*) cuando se accede a una posición con un índice fuera de rango. De esta forma, facilita la detección y corrección de errores de programación, que de otra forma quedarían ocultos, haciendose más difícil su detección y corrección.

```
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
int main()
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    // Detección en tiempo de ejecución de acceso
    // fuera de rango (error condicion <= ),
    // aborta la ejecución del programa
    for (unsigned i = 0; i <= a1.size(); ++i) {
        cout << a1[i] << endl;
    }
}
```

```
const int NELMS = 5;
typedef int APredef[NELMS] ;
int main()
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    // Error de ejecución en acceso
    // fuera de rango (error condicion <= )
    // difícil de detectar y corregir
    for (int i = 0; i <= NELMS; ++i) {
        cout << a1[i] << endl;
    }
}
```

Devolución de un array por parte de de una función

Aunque es una operación costosa, y **se recomienda que NO sea utilizada**, una función sí puede devolver (con la sentencia `return`) un *array* de la biblioteca estándar. Sin embargo, en el caso de los *arrays* predefinidos no es posible (no es consistente con el resto de tipos de C++).

```cpp
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
ABStd valores()              // Correcto
{
    ABStd a1 = {{ 1, 2, 3, 4, 5 }} ;
    return a1 ;              // Correcto
}
```

```cpp
const int NELMS = 5;
typedef int APredef[NELMS] ;
APredef valores()            // ErrorCompilación
{
    APredef a1 = { 1, 2, 3, 4, 5 } ;
    return a1;               // ErrorCompilación
}
```

Paso de parámetros de arrays

Paso de parámetros por valor

Aunque es una operación costosa, y **se recomienda que NO sea utilizada**, sí es posible que un *array* de la biblioteca estándar sea pasado como parámetro *por valor* a un subprograma. Sin embargo, en el caso de los *arrays* predefinidos no es posible (no es consistente con el resto de tipos de C++).

Además, en el caso de pasar como parámetro *por valor* un *array* predefinido, el sistema realiza de forma automática (y sin notificación al programador) un paso como parámetro *por referencia*, lo que puede dar lugar a errores inesperados.

```cpp
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
// Paso por valor: costoso, pero correcto
void sbprg1(ABStd a1)
{
    a1[0] = 20 ;     // Modificacion de copia local
}                    // comportamiento esperado
```

```cpp
const int NELMS = 5;
typedef int APredef[NELMS] ;
// Paso por valor: paso por referencia automático
void sbprg1(APredef a1)
{
    a1[0] = 20 ;     // Modificacion de copia remota
}                    // comportamiento inesperado
```

El paso de parámetros *por valor* de *arrays* predefinidos está sujeto a numerosas peculiaridades que hacen de su tratamiento un caso excepcional con respecto a los otros tipos y es propenso a introducir numerosos errores de programación (especialmente en un contexto de aprendizaje a la programación), por lo que exige un tratamiento cauteloso por parte del programador.

No se comprueba que los parámetros actuales y formales sean del mismo tamaño, *siendo posible* pasar como parámetro actual un array predefinido de un tamaño diferente al especificado como parámetro formal, con los errores que ello conlleva.

```cpp
const int NELMS = 5;
typedef array<int, NELMS> ABStd;

// Paso por valor: costoso, pero correcto
void sbprg(ABStd a1)
{
    for (unsigned i = 0; i < a1.size(); ++i) {
        cout << a1[i] << endl;
    }
}
const int MAX = 3;
typedef array<int, MAX> ATresNum;
int main()
{
    ATresNum a = {{ 1, 2, 3 }};
    sbprg(a) ; // ErrorCompilación
}
```

```cpp
const int NELMS = 5;
typedef int APredef[NELMS] ;
// Paso por valor: paso por referencia automático
// sin comprobación de tamaño
void sbprg(APredef a1)
{
    for (int i = 0; i < NELMS; ++i) {
        cout << a1[i] << endl;
    }
}
const int MAX = 3;
typedef int ATresNum[MAX];
int main()
{
    ATresNum a = { 1, 2, 3 };
    sbprg(a) ; // Error de ejecución no detectable
}
```

Si se utiliza el operador de asignación (=) entre parámetros *por valor* de arrays predefinidos, no se produce ningún error de compilación, aunque obtendremos resultados inesperados. Este comportamiento es diferente si se utiliza la asignación entre arrays predefinidos que no son parámetros, en cuyo caso si se producirá un error de compilación.

```cpp
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
// Paso por valor: costoso, pero correcto
void copiar(ABStd a1, ABStd a2)
{
    a1 = a2 ;   // Asignación de copia local
}               // comportamiento esperado
int main()
{
    ABStd a5 = {{ 1, 2, 3 }};
    ABStd a6;
    copiar(a6, a5) ;
    a6 = a5 ;       // Asignación correcta
}
```

```cpp
const int NELMS = 5;
typedef int APredef[NELMS] ;
// Paso por valor: paso por referencia automático
void copiar(APredef a1, APredef a2)
{
    a1 = a2 ;   // ErrorEjecución, no asigna los elementos
}               // comportamiento inesperado,
int main()
{
    APredef a5 = { 1, 2, 3 };
    APredef a6;
    copiar(a6, a5) ;
    a6 = a5 ;       // ErrorCompilación
}
```

Paso de parámetros por referencia y por referencia constante

Tanto los *arrays* de la biblioteca estándar, como los *arrays* predefinidos, se pueden pasar como parámetros *por referencia* y por *referencia constante*, con numerosas ventajas con respecto al paso por valor.

En el caso de los *arrays* de la biblioteca estándar, aunque el paso de parámetros *por valor* es válido, y con un comportamiento adecuado y consistente, es muy costoso ya que debe duplicar la memoria y copiar todo el contenido del array, el paso de parámetros *por referencia* y *por referencia constante* es muy eficiente y tiene un comportamiento adecuado y consistente.

En el caso de los *arrays* predefinidos, el paso de parámetros *por valor* tiene muchos problemas asociados, que son resueltos adecuadamente por el paso de parámetros *por referencia* y *por referencia constante*, ya que el paso por referencia realiza una comprobación de tipos, y evita los problemas indicados anteriormente que se producían en el paso de parámetros *por valor* de arrays predefinidos.

```cpp
const int NELMS = 5;
typedef array<int, NELMS> ABStd;
void copiar(ABStd& a1, const ABStd& a2)
{
    a1 = a2 ;   // Asignación correcta
    for (unsigned i = 0; i < a1.size(); ++i) {
        a1[i] = a2[i] ;
    }
}
int main()
{
    ABStd a5 = {{ 1, 2, 3 }};
    ABStd a6;
    copiar(a6, a5) ;
    a6 = a5 ;       // Asignación correcta
}
```

```cpp
const int NELMS = 5;
typedef int APredef[NELMS] ;
void copiar(APredef& a1, const APredef& a2)
{
    a1 = a2 ;   // ErrorCompilación
    for (int i = 0; i < NELMS; ++i) {
        a1[i] = a2[i] ;
    }
}
int main()
{
    APredef a5 = { 1, 2, 3 };
    APredef a6;
    copiar(a6, a5) ;
    a6 = a5 ;       // ErrorCompilación
}
```

Conclusiones

Los *arrays* predefinidos son compatibles con el lenguaje de programación C, y por lo tanto son útiles en aquellas circunstancias en las que es adecuado interactuar con bibliotecas de C o del sistema operativo.

Sin embargo, las características de los *arrays* predefinidos no son consistentes con las características proporcionadas por los otros tipos de datos de C++, teniendo muchas particularidades, con características propensas a la introducción de errores, especialmente durante un curso de aprendizaje de la programación.

Por otra parte, las características proporcionadas por los *arrays* de la biblioteca estándar son *consistentes* con las características proporcionadas por los otros tipos de datos de C++, proporcionando un mecanismo *más simple, consistente y robusto* que los *arrays* predefinidos, por lo que son más adecuados para la mayoría de las situaciones.