

Estructura de datos

Nombre: Luis Alejandro Velasco Cantú

Actividad: leer paginas 41 a 67

Usuario de discord: 7215vee#0216

Coreo institucional: luis.velasco@uniat.edu.mx

Fecha de clase: 13/03/2024



Pag 31

Esta página habla de una clases en C++ y su sintaxis básica y dice que:

Las clases en C++: y dice que son estructuras que contienen miembros, que pueden ser datos o funciones. Los miembros de una clase definen su comportamiento y datos asociados. Y los miembros de una clase pueden ser públicos o privados. Los miembros públicos pueden ser accedidos desde cualquier parte del programa, mientras que los privados solo pueden ser accedidos por métodos dentro de la misma clase. Esto permite ocultar detalles de implementación y controlar el acceso a los datos.

Y habla también sobre los constructores son métodos especiales que se utilizan para inicializar una instancia de una clase. Si no se define un constructor, se genera automáticamente uno por defecto. Los constructores pueden aceptar parámetros para inicializar los datos miembros de la clase.

Esta página nos habla y muestra que la clase IntCell en C++, que simula una celda de memoria para enteros. Y nos dice dos constructores: uno sin parámetros, que inicializa el valor en 0, y otro que acepta un valor inicial. La clase también proporciona métodos read y write para leer y escribir en la celda respectivamente. Se utiliza un parámetro predeterminado en el constructor para la inicialización flexible. La encapsulación de datos se garantiza al declarar el valor almacenado como privado.

```
1 /**
2 * A class for simulating an integer memory cell.
3 */
4 class IntCell
5 {
6 public:
7 /**
8 * Construct the IntCell.
9 * Initial value is 0.
10 */
11 IntCell()
12 { storedValue = 0; }
13
14 /**
15 * Construct the IntCell.
16 * Initial value is initialValue.
17 */
18 IntCell( int initialValue )
19 { storedValue = initialValue; }
20
21 /**
22 * Return the stored value.
```

```
23 */
24 int read( )
25 { return storedValue; }

26
27 /**
28 * Change the stored value to x.
29 */
30 void write( int x )
31 { storedValue = x; }

32
33 private:
34 int storedValue;
35 };
```

sta página introduce la clase IntCell en C++, que simula una celda de memoria para enteros. Se presenta un constructor con un parámetro

```
1 /**
2 * A class for simulating an integer memory cell.
3 */
4 class IntCell
5 {
6 public:
7 explicit IntCell( int initialValue = 0 )
8 : storedValue{ initialValue }{ }
9 int read( ) const
10 { return storedValue; }
11 void write( int x )
12 { storedValue = x; }
13
14 private:
15 int storedValue;
16 };
```

Y nos dice que el uso de la lista de inicialización para inicializar directamente los miembros de datos, lo que ahorra tiempo y es especialmente útil para tipos de clase con inicializaciones complejas o miembros de datos const. Además, nos dice la nueva sintaxis de inicialización con llaves en lugar de paréntesis introducida en C++11 para una sintaxis uniforme de inicialización.

Esta pagina habla de dos conceptos importantes en la programación:

el constructor explícito y la función miembro constante.

Constructor explícito: nos recomienda que todos los constructores de un solo parámetro sean explícitos para evitar conversiones de tipo implícitas no deseadas, que pueden llevar a errores difíciles de encontrar. Un constructor explícito impide que se realicen conversiones implícitas al tipo de la clase. Por ejemplo, si un constructor de IntCell es explícito, no se permitirá la asignación obj = 37;, ya que 37 no es del tipo IntCell.

```
IntCell obj; // obj is an IntCell  
obj = 37; // Should not compile: type mismatch
```

Función miembro constante: Una función miembro que no modifica el estado del objeto se conoce como un accessor. En contraste, una función miembro que modifica el estado del objeto se llama mutador. En C++, se puede marcar una función miembro como constante agregando la palabra clave const al final de su declaración. Esto garantiza que la función no modifique el estado del objeto y permite que se llame a la función incluso en objetos constantes. En el ejemplo de la clase IntCell, la función read es un accessor y, por lo tanto, se declara como constante.

la separación de la interfaz y la implementación de una clase en C++.

Separación de interfaz e implementación: En C++, es común separar la interfaz de una clase de su implementación. La interfaz lista la clase y sus miembros (datos y funciones), mientras que la implementación proporciona las implementaciones de las funciones. Y tambien habla sobre la interfaz de la clase IntCell en la Figura 1.7, la implementación en la Figura 1.8 y un programa principal que utiliza IntCell. Se destacan los siguientes puntos:

Comandos del preprocesador: La interfaz se coloca típicamente en un archivo que termina con .h. El código fuente que requiere conocimiento de la interfaz debe incluir el archivo de interfaz mediante el uso de la directiva #include. Para evitar la lectura duplicada de la interfaz en un proyecto complejo, se utiliza el preprocesador para definir un símbolo cuando se lee la interfaz del archivo. Este símbolo se utiliza para evitar la inclusión múltiple del mismo archivo de interfaz en un mismo archivo de código fuente. En el ejemplo, se utiliza #ifndef IntCell_H seguido de #define IntCell_H para mejorar que la interfaz se lea solo una vez durante la compilación.

```
1 ifndef IntCell_H
2 define IntCell_H
3
4 /**
5 * A class for simulating an integer memory cell.
6 */
7 class IntCell
8 {
9 public:
10 explicit IntCell( int initialValue = 0 );
11 int read( ) const;
12 void write( int x );
13
14 private:
```

15 int storedValue;

16 };

17

18 #endif

Pag 36

En esta pagina habla de como poner la clase IntCell y un programa que la utiliza.

Implementación de IntCell: El archivo IntCell.cpp contiene la implementación de la clase IntCell. Define el constructor para inicializar IntCell con un valor inicial, así como las funciones read y write para obtener y modificar el valor almacenado, en su forma mas respectiva.

```
1 #include "IntCell.h"
2
3 /**
4 * Construct the IntCell with initialValue
5 */
6 IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
7 {
8 }
9
10 /**
11 * Return the stored value.
12 */
13 int IntCell::read( ) const
14 {
15     return storedValue;
16 }
17
18 /**
19 * Store x.
20 */
21 void IntCell::write( int x )
22 {
```

```
23 storedValue = x;  
24 }
```

Programa de prueba: El archivo TestIntCell.cpp es un programa de prueba que demuestra cómo utilizar la clase IntCell. En el programa, se crea un objeto IntCell, se escribe un valor en él y luego se lee y muestra el valor almacenado en la consola.

```
1 #include <iostream>  
2 #include "IntCell.h"  
3 using namespace std;  
4  
5 int main( )  
6 {  
7     IntCell m;  
8  
9     m.write( 5 );  
10    cout << "Cell contents: " << m.read( ) << endl;  
11  
12    return 0;  
13 }
```

Pag 37

Esta pagina habla sobre los aspectos prácticos relacionados con la implementación de clases y la utilización de objetos.

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main( )
6 {
7     vector<int> squares( 100 );
8
9     for( int i = 0; i < squares.size( ); ++i )
10    squares[ i ] = i * i;
11
12    for( int i = 0; i < squares.size( ); ++i )
13        cout << i << " " << squares[ i ] << endl;
14
15    return 0;
16 }
```

Directivas del preprocesador: Se utiliza la directiva `#ifndef` junto con `#define` y `#endif` para evitar la inclusión múltiple de archivos de encabezado. Esto ayuda a que un archivo de encabezado solo se procese una vez durante la compilación.

Operador de resolución de ámbito (::): En el archivo de implementación, cada función miembro debe identificar la clase a la que pertenece utilizando el operador de resolución de ámbito. Esto evita que las funciones se asuman en el ámbito global.

Coincidencia exacta de firmas: La firma de una función miembro implementada debe coincidir exactamente con la firma listada en la interfaz de clase. Cualquier diferencia, como la omisión de `const`, resultaría en un error.

Declaración de objetos: Los objetos se declaran de manera similar a los tipos primitivos. Se pueden utilizar constructores con y sin parámetros para inicializar objetos. Pero, se deben evitar las conversiones implícitas de tipos, especialmente cuando se utiliza un constructor explícito.

Inicialización uniforme: la inicialización uniforme utilizando llaves ({}), lo que proporciona una sintaxis más coherente y elimina ambigüedades en la inicialización de objetos con constructores sin parámetros. Esto hace que la sintaxis sea más clara y menos propensa a errores.

```
int daysInMonth[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Pag 38

```
vector<int> daysInMonth( 12 ); // No {} before C++11  
daysInMonth[ 0 ] = 31; daysInMonth[ 1 ] = 28; daysInMonth[ 2 ] = 31;  
daysInMonth[ 3 ] = 30; daysInMonth[ 4 ] = 31; daysInMonth[ 5 ] = 30;  
daysInMonth[ 6 ] = 31; daysInMonth[ 7 ] = 31; daysInMonth[ 8 ] = 30;  
daysInMonth[ 9 ] = 31; daysInMonth[ 10 ] = 30; daysInMonth[ 11 ] = 31;
```

La página habla sobre el uso de la clase vector en C++ y cómo la sintaxis de inicialización ha evolucionado con el estándar C++11. Además, se menciona la utilidad de la clase string y se habla la sintaxis del bucle for de rango, introducida en C++11.

Vector y String: Se menciona que las clases vector y string son clases estándar de C++ que reemplazan las limitaciones de los arreglos integrados de C++. Estas clases permiten una manipulación más flexible y segura de datos, como la capacidad de comparar objetos y copiarlos fácilmente.

Inicialización de Vector: Se destaca que, antes de C++11, inicializar un vector requería asignar elementos individualmente. Con C++11, se introdujo la inicialización uniforme, permitiendo la inicialización del vector con una lista de elementos entre llaves ({}), lo que simplifica la sintaxis.

Ambigüedad de la Sintaxis de Inicialización en C++11: Se discute la ambigüedad en la sintaxis de inicialización del vector con llaves, especialmente en casos como `vector<int> daysInMonth { 12 }`, donde no está claro si se está inicializando un vector de tamaño 12 o un vector con un solo elemento de valor 12. Se menciona que el uso de paréntesis () resuelve esta ambigüedad.

Uso del Bucle for de Rango: Se introduce la sintaxis del bucle for de rango, que simplifica la iteración a través de contenedores, como vectores, sin la necesidad de utilizar índices explícitos. Esto se ilustra con ejemplos de cómo calcular la suma de los elementos de un vector utilizando esta nueva sintaxis. Además, se menciona que la palabra reservada auto se puede utilizar para que el compilador infiera automáticamente el tipo de datos en el bucle de rango.

La página empieza hablando sobre que el bucle for de rango introducido en C++11 y dice que solo esta bien cuando se accede a todos los elementos secuencialmente y no se necesita el índice. Además, se menciona que la modificación de elementos en el bucle de rango requiere una sintaxis diferente.

Luego, se pasa a detallar algunos aspectos de C++:

Nos dice que un puntero es una variable que almacena la dirección de memoria donde reside otro objeto. Y dice que su importancia en la manipulación de estructuras de datos, como las listas enlazadas. Se proporciona un ejemplo que ilustra la declaración y uso de punteros en C++ para realizar asignación dinámica de memoria.

En el ejemplo se dice que un puntero m que apunta a un objeto de tipo IntCell, y se utiliza el operador new para asignar dinámicamente un objeto IntCell. Luego, se accede a los miembros del objeto IntCell utilizando el operador ->. Finalmente, se libera la memoria asignada dinámicamente utilizando delete.

```
1 int main()
2 {
3     IntCell *m;
4
5     m = new IntCell{ 0 };
6     m->write( 5 );
7     cout << "Cell contents: " << m->read() << endl;
8
9     delete m;
10
11 }
```

Al principio hablan **sobre variables no inicializadas**, y nos advierte sobre el peligro de utilizar variables sin inicializar, ya que puede provocar fallos en el programa debido a ubicaciones del sistema que no existen. Y al mismo tiempo nos recomienda asignar un valor inicial a las variables, como nullptr (pd: La palabra clave nullptr representa un valor de puntero nulo. Use un valor de puntero nulo para indicar que un tipo de identificador de objeto, puntero interior o puntero nativo no apunta a un objeto.), para evitar estos problemas.

Luego continua con la **creación dinámica de objetos**: nos muestra cómo crear objetos dinámicamente en C++ utilizando el operador new aquí unos ejemplos:

```
m = new IntCell(); // OK  
m = new IntCell{}; // C++11  
m = new IntCell; // Preferred in this text
```

que devuelve un puntero al objeto recién creado. Se presentan varias formas de crear objetos utilizando su constructor.

Y despues nos habla que C++ no cuenta con **recolección automática de basura**, por lo que tenemos que borrar la memoria de forma dinámicamente utilizando el operador delete cuando los objetos ya no se necesiten. Nos avisan tambien sobre las posibles fugas de memoria (memory leaks) y se dice tambien la importancia de evitar el uso excesivo de new cuando sea posible.

(pd: La administración de la recolección de elementos, que no se llegue a utilizar, es crucial por varias razones de peso. La recolección eficaz de elementos no utilizados ayuda a evitar las fugas de memoria, un problema común en el que la memoria asignada no se desasigna adecuadamente cuando ya no se necesita. Las fugas de memoria pueden provocar una degradación del rendimiento, el agotamiento de recursos y errores imprevistos, lo que hace que la recolección adecuada

La gestión de la recolección de elementos no utilizados ayuda a mitigar la fragmentación de la memoria. Esto ocurre cuando la memoria se asigna y desasigna en diferentes tamaños y patrones, lo que da lugar a bloques de memoria fragmentados.)

Se habla tambien cómo se realiza la **asignación y la comparación** de variables de tipo puntero en C++, basada en la dirección de memoria que almacenan. Mencionan que dos punteros son iguales si apuntan al mismo objeto y nos enseña el uso de la asignación de punteros.

Y por ultimo se nos muestra cómo **acceder a los miembros de un objeto a través de un puntero** utilizando el operador ->.

Al principio de la pagina hablan sobre **operador de dirección (&)** Esto devuelve la dirección de memoria de un objeto. Y sirve para implementar pruebas de alias que son útiles para determinar si dos punteros apuntan al mismo objeto.

Después hablan de un tema llamado **Lvalues, Rvalues y Referencias** empiezan con los conceptos de lvalues y rvalues. Un lvalue se dice que identifica un objeto no temporal, mientras que un rvalue identifica un objeto temporal o un valor sin asociación con ningún objeto. Además, se discuten las referencias a lvalues y rvalues en C++11. A continuación un ejemplo:

```
vector<string> arr( 3 );
const int x = 2;
int y;
...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

con este ejemplo podemos decir que, arr, str, arr[x], &x, y, z, ptr, *ptr, (*ptr)[x] son todos lvalues. Además, x también es un lvalue, aunque no es un lvalue modifiable. Como regla general, si tienes un nombre para una variable, es un lvalue, independientemente de si es modifiable.

Para las declaraciones anteriores, 2, "foo", x+y, str.substr(0,1) son todos rvalues. 2 y "foo" son rvalues porque son literales. Intuitivamente, x+y es un rvalue porque su valor es temporal, ciertamente no es x ni y, pero se almacena en algún lugar antes de ser asignado a z. Una lógica casi igual se aplica a str.substr(0,1).

También se menciona que en C++11 se hablan dos tipos de referencias: las referencias a lvalues y las referencias a rvalues. Estas permiten definir nuevos nombres para valores existentes, con algunas restricciones sobre a qué tipo de objeto pueden referirse.

Inicia hablando de que en C++11, una referencia a lvalue se declara colocando un & después de algún tipo. Una referencia a lvalue luego se convierte en un sinónimo (ósea, otro nombre) para el objeto al que hace referencia, a continuación un ejemplo:

```
string str = "hola";
string & rstr = str; // rstr es otro nombre para str
rstr += 'o'; // cambia str a "hola"
bool cond = (&str == &rstr); // verdadero; str y rstr son el mismo objeto
string & bad1 = "hello"; // ilegal: "hello" no es un lvalue modifiable
string & bad2 = str + ""; // ilegal: str+"" no es un lvalue
string & sub = str.substr(0, 4); // ilegal: str.substr(0, 4) no es un lvalue
```

también habla que C++11, hace una referencia a rvalue y se declara colocando un && después de algún tipo. Una referencia a rvalue es que tiene las mismas características que una referencia a lvalue, excepto que a diferencia de una referencia a lvalue, una referencia a rvalue también puede hacer referencia a un rvalue (en pocas palabras, a un temporario). Por ejemplo:

```
string str = "hola";
string && bad1 = "hello"; // Legal
string && bad2 = str + ""; // Legal
string && sub = str.substr(0, 4); // Legal
```

nos explica el ejemplo que mientras que las referencias a lvalue tienen varios usos claros en C++, la utilidad de las referencias a rvalue no es obvia. Se discutirán varios usos de las referencias a lvalue ahora (las respuestas se muestran en sección 1.5.3)

y más abajo habla del uso de referencias a lvalue #1 “aliasing” de nombres complicados su uso más sencillo de esta será mostrado en el capítulo 5 pero se mostrará el mismo concepto pero más difícil:

```
auto & whichList = theLists[myhash(x, theLists.size())];
if (find(begin(whichList), end(whichList), x) != end(whichList))
    return false;
whichList.push_back(x);
```

En este código se utiliza una variable de referencia para que la expresión considerablemente más compleja `theLists[myhash(x,theLists.size())]` no tenga que escribirse (y luego evaluarse) cuatro veces. Se utiliza este código

```
auto whichList = theLists[myhash(x, theLists.size())];
```

no funcionaría; crearía una copia, y luego la operación `push_back` en la última línea se aplicaría a la copia, no al original.

Pag 44

La pagina comienza con un segundo uso está en la instrucción de rango for. Así que incrementamos en 1 el vector. Esto es fácil con un bucle for y nos muestra este:

```
for (int i = 0; i < arr.size(); ++i)  
    ++arr[i];
```

¿Se podría ver más bonito? Si, pero, el código natural no funciona, porque x asume una copia de cada valor en el vector.

```
for (auto x : arr) // roto  
    ++x;
```

Lo que necesitamos es que x sea otro nombre para cada valor en el vector, lo cual puede serse fácil si x es una referencia:

```
for (auto &x : arr) // funciona  
    ++x;
```

Uso de referencias a lvalue #3: evitando una copia

Aquí se muestra que tenemos una función findMax que devuelve el valor más grande en un vector u otra colección grande. Entonces, dada un vector arr, si invocamos findMax, naturalmente se escribiría:

```
auto x = findMax(arr);
```

y podemos ver que si el vector almacena objetos grandes, entonces el resultado es que x será una copia del valor más grande en arr. Si llegaremos a tener la necesidad una copia por alguna razón, está bien sin embargo, en muchos casos, solo necesitamos el valor y no haremos cambios en x. En tal caso, sería más eficiente declarar que x es otro nombre para el valor más grande en arr, y por lo tanto declararíamos x como una referencia (auto deducirá la constancia; si no se usa auto, entonces típicamente se especifica explícitamente una referencia no modificable con const):

```
auto &x = findMax(arr);
```

Normalmente, esto significa que findMax también especificaría un tipo de retorno que indica una variable de referencia.

Este código nos muestra dos conceptos importantes:

Las variables de referencia se usan a menudo para evitar copiar objetos a través de los límites de llamada a funciones (ya sea en la llamada a la función o en el retorno de la función).

Se necesita sintaxis en las declaraciones y retornos de funciones para permitir el paso y retorno usando referencias en lugar de copias.

Paso de parámetros

Si hay muchos lenguajes, incluidos C y Java, pasan todos los parámetros mediante el paso por valor: el argumento real se copia en el parámetro formal. pero, los parámetros en C++ podrían ser objetos complejos grandes para los cuales la copia es ineficiente. Además, a veces es mejor poder alterar el valor que se pasa. Como resultado de esto, C++ históricamente ha tenido tres formas diferentes de pasar parámetros, y C++11 ha agregado una cuarta. Comenzaremos describiendo los tres mecanismos de paso de parámetros en C++ clásico y luego explicaremos el nuevo mecanismo de paso de parámetros que se ha agregado recientemente.

Para empezar la pagina habla sobre las limitaciones del paso de parámetros por valor en C++ y presenta tres casos de función que ilustran estas limitaciones:

```
double average( double a, double b ); // vuelve average of a and b  
void swap( double a, double b ); // intercambia a and b; equivocados parámetros a b  
string randomItem( vector<string> arr ); // vuelve a un item aleatorio (random item) en arr;  
inefficient
```

average: nos muestra el mejor uso del paso de parámetros por valor, donde los valores de las variables se copian en los parámetros de la función, permitiendo que las variables originales permanezcan inalteradas.

swap: nos muestra por qué el paso de parámetros por valor no es adecuado. Aunque conserva la integridad de las variables originales, no permite intercambiar valores entre las variables de entrada.

randomItem: Expone la inefficiencia del paso de parámetros por valor al pasar estructuras de datos grandes como vectores. En lugar de realizar una costosa operación de copia, se sugiere el uso de referencias constantes para evitar la copia innecesaria.

Para saber estas limitaciones, se utiliza los pasos de parámetros por referencia, permitiendo que los parámetros de las funciones sean referencias a las variables originales. Esta técnica, conocida como "call-by-reference" en C++ clásico y "call-by-lvalue-reference" en C++11, permite modificar directamente las variables originales y evita copias costosas de datos.

En esta página dice que en diferentes mecanismos de paso de parámetros y retorno en C++, junto con las consideraciones para elegir el método correcto:

Paso de Parámetros:

Los pasos de parámetros por referencia se utilizan cuando el parámetro formal debe poder cambiar el valor del argumento actual.

Como por ejemplo para tipos primitivos, se utiliza el paso por valor, para tipos de clase, se recomienda el paso por referencia constante, a menos que el tipo sea pequeño y fácilmente copiable.

En pocas palabras, el paso por valor se utiliza para objetos pequeños que no deben ser alterados, el paso por referencia constante para objetos grandes y costosos de copiar, y el paso por referencia para objetos que pueden ser alterados.

Retorno de Valores:

El retorno por valor es el mecanismo más común, donde la función devuelve un objeto del tipo apropiado que puede ser utilizado por el llamante.

Sin embargo, el retorno por valor puede ser ineficiente para objetos grandes, como en el caso de la función randomItem, donde se recomienda el paso por referencia constante para evitar copias innecesarias.

C++11 agrega referencia a rvalue, hay una cuarta forma de pasar parámetros: llamada por referencia a rvalue. El concepto central es que dado que un rvalue almacena un temporal que está a punto de ser destruido, una expresión como `x=rval` (donde `rval` es un rvalue) puede ser implementada mediante un movimiento en lugar de una copia; a menudo, mover el estado de un objeto es mucho más fácil que copiarlo, ya que puede implicar solo un simple cambio de puntero. Lo que vemos aquí es que `x=y` puede ser una copia si `y` es un lvalue, pero un movimiento si `y` es un rvalue. Esto proporciona un caso de uso primario de sobrecarga de una función según si un parámetro es un lvalue o rvalue, como:

```
string randomItem( const vector<string> & arr ); // vuelve un item random en lvalue arr  
string randomItem( vector<string> && arr ); // vuelve item random en rvalue arr  
  
vector<string> v{ "hello", "world" };  
cout << randomItem( v ) << endl; // invoca lvalue method  
cout << randomItem( { "hello", "world" } ) << endl; // invoca rvalue method
```

el retorno de valores puede ser más eficiente gracias a la optimización de movimiento (en C++11), especialmente en funciones como `partialSum`.

Además que nos menciona que habla sobre el concepto de paso de parámetros por referencia a rvalue en C++11, que aprovecha la optimización de movimiento para mejorar la eficiencia en la manipulación de objetos temporales. Esto nos muestra que mediante ejemplos de sobrecarga de funciones basadas en si un parámetro es un lvalue o un rvalue.

Pag 47

La pagina empieza hablando sobre dos implementaciones de la función randomItem para obtener un elemento aleatorio de un vector. Ejemplo del código que nos muestra:

```
1 LargeType randomItem1( const vector<LargeType> & arr )
2 {
3     return arr[ randomInt( 0, arr.size( ) - 1 ) ];
4 }
5
6 const LargeType & randomItem2( const vector<LargeType> & arr )
7 {
8     return arr[ randomInt( 0, arr.size( ) - 1 ) ];
9 }
10
11 vector<LargeType> vec;
12 ...
13 LargeType item1 = randomItem1( vec ); // copy
14 LargeType item2 = randomItem2( vec ); // copy
15 const LargeType & item3 = randomItem2( vec ); // no copy
```

En una de las dos implementaciones dice que la primera implementación, se devuelve el elemento por valor, lo que significa que una copia del objeto LargeType. Esta copia se realiza debido a que la expresión de retorno podría ser un rvalue, que no existirá lógicamente después de que la llamada a la función termine.

Y la segunda implementación utiliza el retorno por referencia constante para evitar una copia inmediata del objeto LargeType. Sin embargo, el llamante debe usar también una referencia constante para acceder al valor devuelto, de lo contrario, se realizará una copia. Esto se debe a que la referencia constante indica que no se permiten cambios en el valor devuelto.

Además nos menciona que en C++11 se introducen semánticas de movimiento que permiten optimizar el retorno por valor, evitando copias innecesarias. Esto se ilustra en el contexto de

la función `partialSum`, donde la optimización de movimiento permite evitar copias innecesarias de vectores.

La pagina empieza mostrandonos la implementación de una función partialSum que calcula la suma parcial de un vector de enteros y lo devuelve. Después hablan de la implementación que crea un nuevo vector result del mismo tamaño que el vector de entrada arr, y luego calcula la suma parcial acumulativa, almacenándola en result. Finalmente, devuelve el vector result.

```
1 vector<int> partialSum( const vector<int> & arr )
2 {
3     vector<int> result( arr.size( ) );
4
5     result[ 0 ] = arr[ 0 ];
6     for( int i = 1; i < arr.size( ); ++i )
7         result[ i ] = result[ i-1 ] + arr[ i ];
8
9     return result;
10 }
11
12 vector<int> vec;
13 ...
14 vector<int> sums = partialSum( vec ); // copia lo viejo en C++; y lo mueve en C++11
```

Además dicen que la técnica de retorno por referencia, que permite al llamante de una función tener acceso modifiable a la representación de datos interna de una clase.

Y al final hablan sobre la sección también destaca la importancia de std::swap y std::move en C++11 para evitar copias innecesarias y mejorar la eficiencia en la manipulación de objetos. std::move permite convertir cualquier lvalue en un rvalue, lo que nos ayuda a la implementación de funciones como swap para realizar movimientos en lugar de copias. La función std::swap también está disponible en la Biblioteca Estándar y funciona para cualquier tipo.

Pag 49

```
1 void swap( double & x, double & y )
2 {
3     double tmp = x;
4     x = y;
5     y = tmp;
6 }
7
8 void swap( vector<string> & x, vector<string> & y )
9 {
10    vector<string> tmp = x;
11    x = y;
12    y = tmp;
13 }
```

Figure 1.14 Swapping by three copies

```
1 void swap( vector<string> & x, vector<string> & y )
2 {
3     vector<string> tmp = static_cast<vector<string> &&>( x );
4     x = static_cast<vector<string> &&>( y );
5     y = static_cast<vector<string> &&>( tmp );
6 }
7
8 void swap( vector<string> & x, vector<string> & y )
9 {
10    vector<string> tmp = std::move( x );
11    x = std::move( y );
12    y = std::move( tmp );
13 }
```

La página empieza con un código que este tiene dos implementaciones de la función swap, una que utiliza tres copias y otra que utiliza tres movimientos.

En la primera implementación, se realizan tres copias de los objetos x e y, lo que puede ser costoso en términos de rendimiento, especialmente para vectores grandes.

En la segunda implementación, se utilizan movimientos en lugar de copias. Esto se logra utilizando static_cast o std::move para convertir los lvalues x, y y tmp en rvalues, lo que permite mover sus contenidos en lugar de copiarlos.

Además, se habla sobre la importancia de las cinco funciones especiales en C++11: destructor, constructor de copia, constructor de movimiento, operador de asignación de copia y operador de asignación de movimiento, estas funciones, conocidas como "big-five", son proporcionadas por el compilador y se hacen una gestión de recursos y la asignación de memoria para los objetos de una clase. La página también dice que, aunque en muchos casos se puede aceptar el comportamiento predeterminado proporcionado por el compilador para estas funciones, a veces es necesario personalizar su comportamiento.

```
IntCell B = C; // copia la línea si C es lvalue; mueve la linea si C es rvalue
```

```
IntCell B { C }; // copia la línea si C es lvalue; mueve la línea si C es rvalue
```

La pagina habla de dos constructores especiales: el constructor de copia y el constructor de movimiento, que se utilizan para inicializar un nuevo objeto con el mismo estado que otro objeto del mismo tipo.

El constructor de copia se llama cuando se inicializa un nuevo objeto a partir de otro objeto existente que es un lvalue.

El constructor de movimiento se llama cuando se inicializa un nuevo objeto a partir de otro objeto existente que es un rvalue.

Estos dos constructores se utilizan en diferentes situaciones, como la inicialización de objetos, la pasada por valor y el retorno por valor (por defecto), el constructor de copia y el constructor de movimiento copian o mueven cada miembro de datos del objeto en cuestión. Lo mismo se aplica al operador de asignación de copia y al operador de asignación de movimiento, que se utilizan para copiar o mover el estado de un objeto a otro.

Generalmente los valores predeterminados proporcionados por el compilador son suficientes y no es necesario modificarlos.

La página empieza hablando sobre la discusión de los valores predeterminados y las implicaciones en la programación de clases en C++. Nos dice que en la mayoría de los casos, los valores predeterminados proporcionados por el compilador son suficientes y que no es necesario cambiarlos. pero se destaca un problema común que surge cuando una clase contiene un miembro de datos que es un puntero. Este problema se conoce como "copia superficial", que ocurre cuando se copia el valor del puntero en lugar de los objetos apuntados.

```
~IntCell() { cout << "Invoking destructor" << endl; } // Destructor  
IntCell( const IntCell & rhs ) = default; // Copy constructor  
IntCell( IntCell && rhs ) = default; // Move constructor  
IntCell & operator= ( const IntCell & rhs ) = default; // Copy assignment  
IntCell & operator= ( IntCell && rhs ) = default; // Move assignment
```

Para tomar este problema de una "copia profunda" forma bien, acordar bien lo que hace el destructor, el constructor de copia, el constructor de movimiento, el operador de asignación de copia y el operador de asignación de movimiento. Se nos recomienda explícitamente considerar todas estas operaciones al escribir cualquier operación entre las "cinco grandes" de una clase. Nos dice alternativa para definir o deshabilitar explícitamente las operaciones de copia y movimiento.

```
IntCell( const IntCell & rhs ) = delete; // No Copy constructor  
IntCell( IntCell && rhs ) = delete; // No Move constructor  
IntCell & operator= ( const IntCell & rhs ) = delete; // No Copy assignment  
IntCell & operator= ( IntCell && rhs ) = delete; // No Move assignment
```

Pag 52

La página empieza hablando sobre la importancia de implementar manualmente las operaciones de destructor, constructores de copia y movimiento, y operadores de asignación de copia y movimiento cuando los valores predeterminados proporcionados por el compilador no son tan buenos y nos muestra con la clase IntCell, donde se utiliza un puntero para almacenar un valor entero. Nos dice que los problemas comunes surgen cuando los valores de los punteros no se manejan correctamente durante la copia y posiblemente la destrucción de objetos.

```
1 class IntCell
2 {
3 public:
4 explicit IntCell( int initialValue = 0 )
5 { storedValue = new int{ initialValue }; }
6
7 int read( ) const
8 { return *storedValue; }
9 void write( int x )
10 { *storedValue = x; }
11
12 private:
13 int *storedValue;
14 };
```

También nos habla de poner la "copia superficial" lleva a errores de programación y pérdida de memoria, lo que significa que de la "copia profunda" a través de las operaciones personalizadas de la "gran cinco". Nos muestra cómo se pueden implementar estas operaciones correctamente en la clase IntCell para evitar problemas como la copia superficial y también las fugas de memoria.

Pag 53

```

1 int f()
2 {
3 IntCell a{ 2 };
4 IntCell b = a;
5 IntCell c;
6
7 c = b;
8 a.write( 4 );
9 cout << a.read() << endl << b.read() << endl << c.read() << endl;
10
11 return 0;
12 }

```

La página empieza mostrándonos una función donde “f” que realiza operaciones con objetos de la clase IntCell, empezando, crea tres objetos IntCell, a, b, y c, donde a se inicializa con un valor de 2 y b como una copia de a. Luego, asigna el valor de b a c. Después de modificar el valor almacenado en a, imprime los valores almacenados en a, b, y c. La función retorna 0 al final.

```

16 IntCell & operator= ( const IntCell & rhs ) // Copy assignment
17 {
18 IntCell copy = rhs;
19 std::swap( *this, copy );
20 return *this;
21 }

```

Y nos dice que dos implementaciones del operador de asignación de copia (operator=). La primera verifica si se está asignando el objeto a sí mismo y luego copia cada campo de datos según sea necesario. La segunda implementación utiliza el idiomático "copy-and-swap", donde se realiza una copia de rhs, se intercambia con *this, y luego se destruye la copia. Además, muestra que el constructor de movimiento (move constructor), que mueve la representación de datos de rhs a *this, y se asegura de que los datos de rhs estén en un

estado válido pero fácilmente destruible. Y dice que si hay datos no primitivos, estos deben moverse en la lista de inicialización del constructor.

```
IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue }, // Move constructor
```

```
    items{ std::move( rhs.items ) }
```

```
    { rhs.storedValue = nullptr; }
```

Pag 54

```
1 class IntCell
2 {
3 public:
4 explicit IntCell( int initialValue = 0 )
5 { storedValue = new int{ initialValue }; }
6
7 ~IntCell( ) // Destructor
8 { delete storedValue; }
9
10 IntCell( const IntCell & rhs ) // Copy constructor
11 { storedValue = new int{ *rhs.storedValue }; }
12
13 IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue } // Move constructor
14 { rhs.storedValue = nullptr; }
15
16 IntCell & operator= ( const IntCell & rhs ) // Copy assignment
17 {
18 if( this != &rhs )
19 *storedValue = *rhs.storedValue;
20 return *this;
21 }
22
23 IntCell & operator= ( IntCell && rhs ) // Move assignment
24 {
25 std::swap( storedValue, rhs.storedValue );
26 return *this;
27 }
28
```

```
29 int read( ) const
30 { return *storedValue; }
31 void write( int x )
32 { *storedValue = x; }
33
34 private:
35 int *storedValue;
36 };
```

La página empieza hablando de la implementación completa de la clase IntCell, que incluye el constructor, el destructor, el constructor de copia, el constructor de movimiento, el operador de asignación de copia y el operador de asignación de movimiento. La clase almacena un puntero a un entero (`int *storedValue`) y a continuación recordaremos para este punto que hacia cada cosa según el libro.

El constructor crea un nuevo entero en el montón con un valor inicial proporcionado, mientras que el destructor libera la memoria asignada dinámicamente. Los constructores de copia y de movimiento crean nuevos objetos IntCell a partir de otros objetos IntCell, copiando o moviendo el valor del entero almacenado.

Los operadores de asignación de copia y de movimiento se encargan de asignar valores desde otro objeto IntCell. El operador de asignación de copia copia el valor almacenado, mientras que el operador de asignación de movimiento intercambia los punteros a los valores almacenados.

Se destaca que el operador de asignación de movimiento se implementa como un intercambio de miembro a miembro. Aunque a veces se implementa como un solo intercambio de objetos de la misma manera que el operador de asignación de copia, esto solo funciona si el intercambio en sí se implementa como un intercambio de miembro a miembro. Si el intercambio se implementa como tres movimientos, entonces tendríamos una recursión mutua no terminante.

Pag 55

La página empieza hablando del tipo de arreglo incorporado en C++, que es importante en un puntero a memoria con un tamaño específico. Se dice que al asignar un arreglo no copia el arreglo en sí, sino que copia solo el puntero a la memoria. Esto significa que la información sobre el tamaño del arreglo se pierde cuando se pasa a una función, por lo que el tamaño del arreglo debe pasarse como un parámetro adicional.

Luego, nos dice que la creación de arreglos dinámicos utilizando `new[]` y cómo estos deben liberarse con `delete[]` para evitar pérdidas de memoria.

Y también nos dicen que las cadenas de estilo C, que son esencialmente arreglos de caracteres terminados por el carácter nulo \0. tambien dice que sobre los problemas asociados con la manipulación de este tipo de arreglos, como la dificultad de la gestión de memoria y los errores potenciales al copiar cadenas.

Se dice que las clases estándar `vector` y `string` en C++ están diseñadas para manejar estos problemas y proporcionar una interfaz más segura y fácil de usar. pero, se sabe que a veces es necesario trabajar con arreglos de estilo C, especialmente cuando se utiliza con bibliotecas diseñadas para trabajar tanto con C como con C++.

La pagina termina con el concepto de plantillas en C++, que permite escribir algoritmos y estructuras de datos que son independientes del tipo de datos con los que trabajan. Esto nos ayuda para escribir código reutilizable que pueda adaptarse a diferentes tipos de datos sin necesidad de recodificarlo para cada tipo.

En esta pagina nos dicen que los algoritmos independientes del tipo C++ utilizando plantillas. Comienza con las plantillas de función, que son patrones para lo que podría ser una función. Por así decirlo la plantilla de función `findMax` muestra cómo se puede escribir un algoritmo genérico para encontrar el máximo de un vector de cualquier tipo de datos comparable.

Las plantillas de función se expanden automáticamente según sea necesario, generando código adicional para cada nuevo tipo utilizado como argumento de plantilla. Sin embargo, pueden surgir errores de compilación si no se cumplen ciertas restricciones en los tipos de datos utilizados con la plantilla.

Y nos dice que la importancia de documentar claramente las asunciones sobre los argumentos de las plantillas para evitar errores y se dice que el uso de convenciones de paso de parámetros y retorno que consideren que los argumentos de plantilla pueden ser tipos de clase en lugar de tipos primitivos.

Y por último, nos dice que las reglas y problemas comunes relacionados con las plantillas de función, especialmente cuando no se puede encontrar una coincidencia exacta para los parámetros de la plantilla.

```
1 /**
2 * Return the maximum item in array a.
3 * Assumes a.size( ) > 0.
4 * Comparable objects must provide operator< and operator=
5 */
6 template <typename Comparable>
7 const Comparable & findMax( const vector<Comparable> & a )
8 {
9 int maxIndex = 0;
10
11 for( int i = 1; i < a.size( ); ++i )
12 if( a[ maxIndex ] < a[i] )
13 maxIndex = i;
```

14 return a[maxIndex];

15 }

Pag 57

```
1 int main()
2 {
3     vector<int> v1( 37 );
4     vector<double> v2( 40 );
5     vector<string> v3( 80 );
6     vector<IntCell> v4( 75 );
7
8 // Additional code to fill in the vectors not shown
9
10 cout << findMax( v1 ) << endl; // OK: Comparable = int
11 cout << findMax( v2 ) << endl; // OK: Comparable = double
12 cout << findMax( v3 ) << endl; // OK: Comparable = string
13 cout << findMax( v4 ) << endl; // Illegal: operator< undefined
14
15 return 0;
16 }
```

Este código muestra el uso del template de función `findMax` con diferentes tipos de vectores. Se crean varios vectores de diferentes tipos, incluyendo `int`, `double`, `string` y `IntCell`, y se llama a `findMax` con cada uno de ellos.

La función `findMax` se correctamente a los vectores de tipos `int`, `double` y `string`, ya que estos tipos son comparables y la función puede encontrar el máximo utilizando el operador de comparación `<`. Sin embargo, al intentar llamar a `findMax` con un vector de tipo `IntCell`, se produce un error de compilación porque el operador `<` no está definido para el tipo `IntCell`.

Nos dice que las reglas para resolver ambigüedades en las plantillas de función son complejas y que si hay una plantilla y una función no plantilla que coinciden, la función no plantilla tiene prioridad. Además, si hay dos coincidencias aproximadas igualmente cercanas, el código es ilegal y el compilador declarará una ambigüedad.

```
1 /**
2 * A class for simulating a memory cell.
3 */
4 template <typename Object>
5 class MemoryCell
6 {
7 public:
8 explicit MemoryCell( const Object & initialValue = Object{ } )
9 : storedValue{ initialValue }{ }
10 const Object & read( ) const
11 { return storedValue; }
12 void write( const Object & x )
13 { storedValue = x; }
14 private:
15 Object storedValue;
16};
```

Luego, se introduce el concepto de plantillas de clase con un ejemplo simple de la plantilla MemoryCell, que funciona como una celda de memoria para almacenar cualquier tipo de dato. La plantilla MemoryCell se declara utilizando la palabra clave template y se define con un tipo de objeto genérico Object.

Pag 58

```
1 int main()
2 {
3     MemoryCell<int> m1;
4     MemoryCell<string> m2{ "hello" };
5
6     m1.write( 37 );
7     m2.write( m2.read() + "world" );
8     cout << m1.read() << endl << m2.read() << endl;
9
10    return 0;
11 }
```

Este código muestra el uso de la plantilla de clase `MemoryCell`. Se crean dos instancias de `MemoryCell`, una para almacenar valores de tipo `int` y otra para almacenar valores de tipo `string`. Se escribe un valor en cada instancia utilizando el método `write`, y luego se lee el valor utilizando el método `read`. Finalmente, los valores leídos se imprimen en la consola.

La plantilla de clase `MemoryCell` permite almacenar objetos de cualquier tipo `Object`, siempre que `Object` tenga un constructor sin parámetros, un constructor de copia y un operador de asignación de copia y nos destaca que `Object` se pasa por referencia constante. Además, el parámetro predeterminado para el constructor no es 0, ya que 0 podría no ser un valor válido para `Object`. En su lugar, el parámetro predeterminado es el resultado de construir un objeto con su constructor sin parámetros.

Y también nos dice que `MemoryCell` no es una clase en sí misma, sino una plantilla de clase, y que `MemoryCell<int>` y `MemoryCell<string>` son las clases reales instanciadas a partir de la plantilla. Ya que además, se discute brevemente la implementación de las plantillas de clase, mencionando que a veces se implementan como una sola unidad debido a problemas de compilación en plataformas que no manejan bien la compilación separada de plantillas.

Y por ultimo, se hace referencia al uso repetido de los tipos genéricos Object y Comparable en el texto, donde Object se asume que tiene ciertas características como constructor sin parámetros y operador de asignación, mientras que Comparable se usa en contextos donde se necesita una ordenación total, como en el ejemplo de findMax.

Pag 59

```
1 class Square
2 {
3 public:
4 explicit Square( double s = 0.0 ) : side{ s }
5 {}
6
7 double getSide( ) const
8 { return side; }
9 double getArea( ) const
10 { return side * side; }
11 double getPerimeter( ) const
12 { return 4 * side; }
13
14 void print( ostream & out = cout ) const
15 { out << "(square " << getSide( ) << ")"; }
16 bool operator< ( const Square & rhs ) const
17 { return getSide( ) < rhs.getSide( ); }
18
19 private:
20 double side;
21 };
22
23 // Define an output operator for Square
24 ostream & operator<< ( ostream & out, const Square & rhs )
25 {
26 rhs.print( out );
27 return out;
28 }
```

```
29
30 int main( )
31 {
32 vector<Square> v = { Square{ 3.0 }, Square{ 2.0 }, Square{ 2.5 } };
33
34 cout << "Largest square: " << findMax( v ) << endl;
35
36 return 0;
37 }
```

El código nos muestra una clase llamada `Square` que representa un cuadrado almacenando la longitud de un lado. La clase proporciona varios métodos para obtener información sobre el cuadrado, como el área y el perímetro. Además, la clase sobrecarga el operador `<`, lo que permite comparar dos objetos de tipo `Square` basándose en la longitud de sus lados.

En el programa principal, se crea un vector de objetos `Square` y se utiliza la función `findMax` para encontrar el cuadrado más grande en el vector. Esto muestra cómo la clase `Square` puede utilizarse como un tipo `Comparable` en el contexto de la función `findMax`.

El código también habla sobre una implementación de la función `print`, que imprime la información del cuadrado en la consola. Esta función se utiliza dentro de la sobrecarga del operador `<<`, que permite imprimir objetos de tipo `Square` de manera conveniente

Pag 60

En esta pagina habla de cómo proporcionar una función de salida para un nuevo tipo de clase, utilizando una función miembro pública llamada print, que toma un ostream como parámetro. Esta función pública puede ser llamada por una función global no perteneciente a la clase, operator<<, que acepta un ostream y un objeto para imprimirla.

Luego, dice el concepto de "objetos de función" en C++, que son utilizados para superar las limitaciones de los templates de función al trabajar con objetos que no tienen un operador < definido. Proporciona una solución al permitir pasar una función de comparación como parámetro, lo que Desa cómoda completamente la lógica de comparación de los objetos en el array.

La idea de como se muestra en el código, donde findMax toma un segundo parámetro que es un tipo genérico. Para que el template de findMax se expanda correctamente, el tipo genérico debe tener una función miembro llamada isLessThan, que toma dos parámetros del primer tipo genérico y devuelve un booleano.

Se menciona que en C++, los objetos de función se implementan utilizando esta idea básica pero con una sintaxis más compleja, utilizando la sobrecarga del operador () en lugar de una función con nombre.

Pag 61

```
1 // Generic findMax, with a function object, Version #1.  
2 // Precondition: a.size( ) > 0.  
3 template <typename Object, typename Comparator>  
4 const Object & findMax( const vector<Object> & arr, Comparator cmp )  
5 {  
6     int maxIndex = 0;  
7  
8     for( int i = 1; i < arr.size( ); ++i )  
9         if( cmp.isLessThan( arr[ maxIndex ], arr[ i ] ) )  
10            maxIndex = i;  
11  
12    return arr[ maxIndex ];  
13 }  
14  
15 class CaseInsensitiveCompare  
16 {  
17 public:  
18     bool isLessThan( const string & lhs, const string & rhs ) const  
19     { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }  
20 };  
21  
22 int main( )  
23 {  
24     vector<string> arr = { "ZEBRA", "alligator", "crocodile" };  
25     cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;  
26  
27     return 0;  
28 }
```

El código nos habla de una versión genérica de `findMax`, que toma un vector de objetos y un objeto de función como parámetros. La función de comparación `isLessThan` del objeto de función se utiliza para determinar el máximo elemento en el vector. En el ejemplo dado, se utiliza un objeto de función `CaseInsensitiveCompare` para comparar cadenas de texto de manera que ignore las diferencias de mayúsculas y minúsculas.

Se dice que el operador () se sobrecarga para permitir la llamada de función, lo que permite usar `cmp(x,y)` en lugar de `cmp.isLessThan(x,y)`. Se proporciona una versión alternativa de `findMax` que utiliza el objeto de función `less` de la biblioteca estándar para realizar comparaciones por defecto.

Además, se adelanta que en el Capítulo 4 se presentará un ejemplo de una clase que necesita ordenar los elementos que almacena, y se mostrará cómo ajustar el código para usar objetos de función.

Pag 62 y 63

```
1 // Generic findMax, with a function object, C++ style.  
2 // Precondition: a.size( ) > 0.  
3 template <typename Object, typename Comparator>  
4 const Object & findMax( const vector<Object> & arr, Comparator isLessThan )  
5 {  
6     int maxIndex = 0;  
7  
8     for( int i = 1; i < arr.size( ); ++i )  
9         if( isLessThan( arr[ maxIndex ], arr[ i ] ))  
10            maxIndex = i;  
11  
12    return arr[ maxIndex ];  
13 }  
14  
15 // Generic findMax, using default ordering.  
16 #include <functional>  
17 template <typename Object>  
18 const Object & findMax( const vector<Object> & arr )  
19 {  
20     return findMax( arr, less<Object>{ } );  
21 }  
22  
23 class CaseInsensitiveCompare  
24 {  
25     public:  
26         bool operator( )( const string & lhs, const string & rhs ) const  
27         { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
```

```
28 };
29
30 int main( )
31 {
32 vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
33
34 cout << findMax( arr, CaselnsensitiveCompare{ } ) << endl;
35 cout << findMax( arr ) << endl;
36
37 return 0;
38 }
```

La primera página en resumen dice que el uso de objetos de función en C++, mostrando una segunda versión de `findMax` que utiliza un objeto de función para realizar comparaciones. En el ejemplo mostrado, se utiliza un objeto de función `CaselnsensitiveCompare` para comparar cadenas de texto sin distinguir entre mayúsculas y minúsculas. Esto se demuestra en el contexto de encontrar el máximo elemento en un vector de cadenas.

La segunda página habla de la compilación por separado de plantillas de clases. Y dice que, históricamente, el soporte del compilador para la compilación por separado de plantillas ha sido débil y específico de la plataforma. Y que, en muchos casos, toda la plantilla de clase con su implementación se coloca en un solo archivo de encabezado. Se dice que, aunque la interfaz de una plantilla de clase se puede declarar de manera similar a una clase regular, la implementación puede ser complicada y propensa a problemas de compilación. Por esta razón, en el código en línea que acompaña al texto, todas las plantillas de clase se implementan completamente en su declaración en un solo archivo de encabezado. La página también menciona que se proporciona información sobre cómo realizar la compilación por separado de plantillas en el apéndice A.

Y por ultimo, la página presenta el concepto de matrices en C++, explicando que la biblioteca estándar de C++ no da una clase de matriz, pero se puede escribir rápidamente una clase de matriz utilizando un vector de vectores. Se proporciona una implementación básica de la clase de matriz, junto con una descripción de sus miembros de datos, constructor y accesores básicos.

Pag 64 y 65

```
1 #ifndef MATRIX_H
2 #define MATRIX_H
3
4 #include <vector>
5 using namespace std;
6
7 template <typename Object>
8 class matrix
9 {
10 public:
11 matrix( int rows, int cols ) : array( rows )
12 {
13 for( auto & thisRow : array )
14 thisRow.resize( cols );
15 }
16
17 matrix( vector<vector<Object>> v ) : array{ v }
18 {}
19 matrix( vector<vector<Object>> &&v) : array{ std::move( v ) }
20 {}
21
22 const vector<Object> & operator[]( int row ) const
23 { return array[ row ]; }
24 vector<Object> & operator[]( int row )
25 { return array[ row ]; }
26
27 int numrows( ) const
28 { return array.size( ); }
```

```
29 int numcols( ) const
30 { return numrows( ) ? array[ 0 ].size( ) : 0; }
31 private:
32 vector<vector<Object>> array;
33 };
34 #endif
```

La página comienza mostrándonos la implementación de la clase matrix, que muestra una matriz mediante un vector de vectores. Se proporcionan tres constructores: uno que inicializa la matriz con un número dado de filas y columnas, otro que inicializa la matriz a partir de un vector de vectores y un tercero que utiliza el operador de movimiento para evitar copias innecesarias de datos.

Luego se habla la implementación del operador [], que permite acceder a las filas de la matriz. Se argumenta sobre si el operador [] debe devolver una referencia constante o una referencia simple, concluyendo que se necesitan dos versiones del operador, una para matrices de solo lectura y otra para matrices modificables.

Y por ultimo nos dice que las funciones de copia, movimiento, destrucción y asignación son manejadas automáticamente por la clase vector, por lo que no es necesario implementarlas explícitamente para la clase de matriz. Se concluye el libro con la sección diciendo que el tiempo de ejecución de un algoritmo frente a grandes cantidades de entrada es un criterio importante para determinar su eficacia, y que se abordarán estas cuestiones en el próximo capítulo utilizando la base matemática discutida en este capítulo.

En la primera pagina nos presentan problemas y la segunda pagina nos empiezan hablando sobre referencias a continuación les diremos todas las referencias que el libro dice:

1. M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, 1988.
2. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Co., Reston, Va., 1982.
3. R. A. Brualdi, *Introductory Combinatorics*, 5th ed., Pearson, Boston, Mass., 2009.
4. E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.
5. B. Eckel, *Thinking in C++*, 2d ed., Prentice Hall, Englewood Cliffs, N.J., 2002.
6. R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass., 1989.
7. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

References 49

8. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2d ed., McGraw-Hill, New York, 1978.
9. D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, 3d ed., Addison-Wesley, Reading, Mass., 1997.
10. S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*, 5th ed., Pearson, Boston, Mass., 2013.
11. S. Meyers, *50 Specific Ways to Improve Your Programs and Designs*, 3d ed., Addison-Wesley, Boston, Mass., 2005.
12. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, Mass., 1996.

13. D. R. Musser, G. J. Durge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming*
with the Standard Template Library, 2d ed., Addison-Wesley, Reading, Mass., 2001.
 14. F. S. Roberts and B. Tesman, *Applied Combinatorics*, 2d ed., Prentice Hall, Englewood Cliffs, N.J., 2003.
 15. B. Stroustrup, *The C++ Programming Language*, 4th ed., Pearson, Boston, Mass., 2013.
 16. A. Tucker, *Applied Combinatorics*, 6th ed., John Wiley & Sons, New York, 2012.
-
17. M. A. Weiss, *Algorithms, Data Structures, and Problem Solving with C++*, 2nd ed., Addison-Wesley, Reading, Mass., 2000.

Referencias que yo busque o utilize:

- [1] TylerMSFT, “*nullptr (C++/CLI y C++/CX)*,” *Microsoft Learn*, Jun. 16, 2023.
<https://learn.microsoft.com/es-es/cpp/extensions/nullptr-cpp-component-extensions?view=msvc-170>
- [2] “*Data Structures & Algorithm Analysis* Mark Allen Weiss.”
<https://classroom.google.com/c/NjY2MjkwMzg0ODgz/m/NjY3NzEyMzc1MDM1/details>
- [3] “*How can you manage garbage collection in C++?*,” [www.linkedin.com](https://www.linkedin.com/advice/1/how-can-you-manage-garbage-collection-c-skills-computer-science-p3i0c#:~:text=Managing%20garbage%20collection%20manually%20in,no%20longer%20needed%2C%20use%20delete.), Jan. 15, 2024.
<https://www.linkedin.com/advice/1/how-can-you-manage-garbage-collection-c-skills-computer-science-p3i0c#:~:text=Managing%20garbage%20collection%20manually%20in,no%20longer%20needed%2C%20use%20delete.>
- [4] TylerMSFT, “*Categorías de valor: Lvalues y Rvalues (C++)*,” *Microsoft Learn*, Jan. 10, 2024. <https://learn.microsoft.com/es-es/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=msvc-170>

Conclusión

Pues podemos concluir que vimos muchas cosas importantes en este capítulo tanto como los grandes 5, garbage, estructuras de datos, diseños de clases y que generalmente estos cosas son esenciales para hacer un buen software sólido y eficiente en C++