1. **Investigar y describir sobre Move Semantics, lvalue y rvalue**

**Move Semantics**

Las referencias de Rvalue admiten la implementación de la semántica de movimiento, que puede aumentar significativamente el rendimiento de sus aplicaciones. La semántica de movimiento le permite escribir código que transfiere recursos (como la memoria asignada dinámicamente) de un objeto a otro. La semántica de movimiento funciona porque permite transferir recursos de objetos temporales a los que no se puede hacer referencia en otro lugar del programa.

Para implementar la semántica de movimiento, normalmente proporciona un constructor de movimiento y, opcionalmente, un operador de asignación de movimiento (operador =) a su clase. Las operaciones de copia y asignación cuyas fuentes son valores se aprovechan automáticamente de la semántica de movimiento. A diferencia del constructor de copia predeterminado, el compilador no proporciona un constructor de movimiento predeterminado. Para obtener más información acerca de cómo escribir un constructor de movimiento y cómo usarlo en su aplicación, vea Constructores de movimiento y Operadores de asignación de movimiento (C ++).

También puede sobrecargar funciones y operadores normales para aprovechar la semántica de movimiento. Visual Studio 2010 introduce la semántica de movimiento en la Biblioteca estándar de C ++. Por ejemplo, la clase de cadena implementa operaciones que realizan semántica de movimiento. Considere el siguiente ejemplo que concatena varias cadenas e imprime el resultado:

// string\_concatenation.cpp

// compile with: /EHsc

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s = string("h") + "e" + "ll" + "o";

cout << s << endl;

}

***You can overload a function to take an lvalue reference and an rvalue reference.***

Al sobrecargar una función para tomar una referencia de valor constante o una referencia de valor, puede escribir código que distinga entre objetos no modificables (valores) y valores temporales modificables (valores). Puede pasar un objeto a una función que tome una referencia de valor r a menos que el objeto esté marcado como constante. El siguiente ejemplo muestra la función f, que está sobrecargada para tomar una referencia de valor y una referencia de valor. La función principal llama a f con ambos valores y un valor r.

// reference-overload.cpp

// Compile with: /EHsc

#include <iostream>

using namespace std;

// A class that contains a memory resource.

class MemoryBlock

{

// TODO: Add resources for the class here.

};

void f(const MemoryBlock&)

{

cout << "In f(const MemoryBlock&). This version cannot modify the parameter." << endl;

}

void f(MemoryBlock&&)

{

cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;

}

int main()

{

MemoryBlock block;

f(block);

f(MemoryBlock());

}

1. **Implementar y explicar un ejemplo explicando Move Semantics, lvalue y rvalue**

// cast-reference.cpp

// Compile with: /EHsc

#include <iostream>

using namespace std;

// A class that contains a memory resource.

class MemoryBlock

{

// TODO: Add resources for the class here.

};

void g(const MemoryBlock&)

{

cout << "In g(const MemoryBlock&)." << endl;

}

void g(MemoryBlock&&)

{

cout << "In g(MemoryBlock&&)." << endl;

}

int main()

{

MemoryBlock block;

g(block);

g(static\_cast<MemoryBlock&&>(block));

}

1. **Investigar y describir sobre lvalue y referencia rvalue**

Cuando escribe una función que toma una referencia rvalue como su parámetro, ese parámetro se trata como un lvalue en el cuerpo de la función. El compilador trata una referencia de rvalue con nombre como un lvalue porque varias partes de un programa pueden hacer referencia a un objeto con nombre; Sería peligroso permitir que múltiples partes de un programa modifiquen o eliminen recursos de ese objeto. Por ejemplo, si varias partes de un programa intentan transferir recursos del mismo objeto, solo la primera parte transferirá con éxito el recurso.

El siguiente ejemplo muestra la función g, que está sobrecargada para tomar una referencia de valor y una referencia de valor. La función f toma una referencia rvalue como parámetro (una referencia rvalue con nombre) y devuelve una referencia rvalue (una referencia rvalue sin nombre). En la llamada a g desde f, la resolución de sobrecarga selecciona la versión de g que toma una referencia de lvalue porque el cuerpo de f trata su parámetro como un lvalue. En la llamada a g desde main, la resolución de sobrecarga selecciona la versión de g que toma una referencia rvalue porque f devuelve una referencia rvalue.

1. **Implementar y explicar un ejemplo explicando lvalue y referencia rvalue**

// named-reference.cpp

// Compile with: /EHsc

#include <iostream>

using namespace std;

// A class that contains a memory resource.

class MemoryBlock

{

// TODO: Add resources for the class here.

};

void g(const MemoryBlock&)

{

cout << "In g(const MemoryBlock&)." << endl;

}

void g(MemoryBlock&&)

{

cout << "In g(MemoryBlock&&)." << endl;

}

MemoryBlock&& f(MemoryBlock&& block)

{

g(block);

return move(block);

}

int main()

{

g(f(MemoryBlock()));

}

This example produces the following output:

C++Copy

In g(const MemoryBlock&).

In g(MemoryBlock&&).

1. **Investigar, describir e implementar un ejemplo usando std::move**

La función **std :: move** de la Biblioteca estándar de C ++ le permite convertir un objeto en una referencia de valor r a ese objeto. Alternativamente, puede usar la palabra clave static\_cast para convertir un valor de l a una referencia de valor de r, como se muestra en el siguiente ejemplo:

// cast-reference.cpp

// Compile with: /EHsc

#include <iostream>

using namespace std;

// A class that contains a memory resource.

class MemoryBlock

{

// TODO: Add resources for the class here.

};

void g(const MemoryBlock&)

{

cout << "In g(const MemoryBlock&)." << endl;

}

void g(MemoryBlock&&)

{

cout << "In g(MemoryBlock&&)." << endl;

}

int main()

{

MemoryBlock block;

g(block);

g(static\_cast<MemoryBlock&&>(block));

}

1. **Investigar, describir e implementar un ejemplo usando el contructor move (move constructor)**

Este tema describe cómo escribir un constructor de movimientos y un operador de asignación de movimientos para una clase C ++. Un constructor de movimiento permite que los recursos que posee un objeto rvalue se muevan a un valor lva sin copiar. Para obtener más información sobre la semántica de movimiento, consulte Rvalue Reference Declarator: &&.

// MemoryBlock.h

#pragma once

#include <iostream>

#include <algorithm>

class MemoryBlock

{

public:

// Simple constructor that initializes the resource.

explicit MemoryBlock(size\_t length)

: \_length(length)

, \_data(new int[length])

{

std::cout << "In MemoryBlock(size\_t). length = "

<< \_length << "." << std::endl;

}

// Destructor.

~MemoryBlock()

{

std::cout << "In ~MemoryBlock(). length = "

<< \_length << ".";

if (\_data != nullptr)

{

std::cout << " Deleting resource.";

// Delete the resource.

delete[] \_data;

}

std::cout << std::endl;

}

// Copy constructor.

MemoryBlock(const MemoryBlock& other)

: \_length(other.\_length)

, \_data(new int[other.\_length])

{

std::cout << "In MemoryBlock(const MemoryBlock&). length = "

<< other.\_length << ". Copying resource." << std::endl;

std::copy(other.\_data, other.\_data + \_length, \_data);

}

// Copy assignment operator.

MemoryBlock& operator=(const MemoryBlock& other)

{

std::cout << "In operator=(const MemoryBlock&). length = "

<< other.\_length << ". Copying resource." << std::endl;

if (this != &other)

{

// Free the existing resource.

delete[] \_data;

\_length = other.\_length;

\_data = new int[\_length];

std::copy(other.\_data, other.\_data + \_length, \_data);

}

return \*this;

}

// Retrieves the length of the data resource.

size\_t Length() const

{

return \_length;

}

private:

size\_t \_length; // The length of the resource.

int\* \_data; // The resource.

};

1. **Investigar, describir e implementar un ejemplo usando el operador de asignación move (move assigment constructor)**

El siguiente ejemplo muestra el constructor de movimiento completo y el operador de asignación de movimiento para la clase MemoryBlock:

// Move constructor.

MemoryBlock(MemoryBlock&& other) noexcept

: \_data(nullptr)

, \_length(0)

{

std::cout << "In MemoryBlock(MemoryBlock&&). length = "

<< other.\_length << ". Moving resource." << std::endl;

// Copy the data pointer and its length from the

// source object.

\_data = other.\_data;

\_length = other.\_length;

// Release the data pointer from the source object so that

// the destructor does not free the memory multiple times.

other.\_data = nullptr;

other.\_length = 0;

}

// Move assignment operator.

MemoryBlock& operator=(MemoryBlock&& other) noexcept

{

std::cout << "In operator=(MemoryBlock&&). length = "

<< other.\_length << "." << std::endl;

if (this != &other)

{

// Free the existing resource.

delete[] \_data;

// Copy the data pointer and its length from the

// source object.

\_data = other.\_data;

\_length = other.\_length;

// Release the data pointer from the source object so that

// the destructor does not free the memory multiple times.

other.\_data = nullptr;

other.\_length = 0;

}

return \*this;

}