Пример 10.01. Приведение типов в С++.

# include <iostream>

using namespace std;

class A

{

int a = 0;

public:

virtual ~A() = 0;

void f() { cout << "method f class A:" << a << endl; }

};

A::~A() {}

class B : public A

{

int b = 1;

public:

void f() { cout << "method f class B;" << b << endl; }

void g1() { cout << "method g1 class B;" << endl; }

};

class C : public B

{

int c = 2;

public:

void f() { cout << "method f class C;" << c << endl; }

void g2() { cout << "method g2 class B;" << endl; }

};

class D : public A

{

int d = 3;

public:

void f() { cout << "method f class D;" << d << endl; }

};

int main()

{

A\* pa = new B;

B\* pb = static\_cast<B\*>(pa);

pb->f();

C\* pc = static\_cast<C\*>(pa);

pc->f();

D\* pd = static\_cast<D\*>(pa);

pd->f();

pb = dynamic\_cast<B\*>(pa);

if (!pb)

{

cout << "Error bad cast!" << endl;

}

else

{

pb->f();

pb->g1();

}

pc = dynamic\_cast<C\*>(pa);

if (!pc)

{

cout << "Error bad cast!" << endl;

}

else

{

pc->f();

pc->g2();

}

const B obj;

const B\* p = &obj;

const\_cast<B\*>(p)->f();

}

Пример 10.02. Реализация хранителя unique\_ptr.

# include <iostream>

using namespace std;

template <typename Type>

class UniquePtr

{

public:

UniquePtr() = default;

constexpr UniquePtr(nullptr\_t) {}

explicit UniquePtr(Type\* p) noexcept : ptr(p) {}

UniquePtr(UniquePtr<Type>&& vright) noexcept;

~UniquePtr() { delete ptr; }

UniquePtr<Type>& operator=(nullptr\_t) noexcept;

UniquePtr<Type>& operator=(UniquePtr<Type>&& vright) noexcept;

Type& operator\*() const noexcept { return \*ptr; }

Type\* const operator->() const noexcept { return ptr; }

explicit operator bool() const noexcept { return ptr != nullptr; }

Type\* get() const noexcept { return ptr; }

Type\* release() noexcept;

void reset(Type\* p = nullptr) noexcept;

UniquePtr(const UniquePtr<Type>&) = delete;

UniquePtr& operator=(const UniquePtr<Type>&) = delete;

private:

Type\* ptr{ nullptr };

};

#pragma region Method UniquePtr

template <typename Type>

UniquePtr<Type>::UniquePtr(UniquePtr<Type>&& vright) noexcept

{

ptr = vright.ptr;

vright.ptr = nullptr;

}

template <typename Type>

UniquePtr<Type>& UniquePtr<Type>::operator=(nullptr\_t) noexcept

{

reset();

return \*this;

}

template <typename Type>

UniquePtr<Type>& UniquePtr<Type>::operator=(UniquePtr<Type>&& vright) noexcept

{

ptr = vright.ptr;

vright.ptr = nullptr;

return \*this;

}

template <typename Type>

Type\* UniquePtr<Type>::release() noexcept

{

Type\* p = ptr;

ptr = nullptr;

return p;

}

template <typename Type>

void UniquePtr<Type>::reset(Type\* p) noexcept

{

delete ptr;

ptr = p;

}

namespace Unique

{

template <typename Type>

UniquePtr<Type> move(const UniquePtr<Type>& unique)

{

return UniquePtr<Type>(const\_cast<UniquePtr<Type>&>(unique).release());

}

template <typename Type, typename... Args>

UniquePtr<Type> make(Args&&... args)

{

return UniquePtr<Type>(new Type(std::forward<Args>(args)...));

}

}

#pragma endregion

class A

{

int key;

public:

A(int k) : key(k)

{

cout << "Calling the constructor of class A (obj" << key << ");" << endl;

}

~A()

{

cout << "Calling a class A destructor (obj" << key << ");" << endl;

}

void f() { cout << "Method f;" << endl; }

};

int main()

{

UniquePtr<A> obj1(new A(1));

UniquePtr<A> obj2 = Unique::make<A>(2);

obj1->f();

(\*obj1).f();

UniquePtr<A> obj3;

// obj3 = obj1; // Error!!!

if(!obj3)

obj3 = Unique::move(obj1);

}

Пример 10.03. Реализация shared\_ptr и weak\_ptr.

# include <iostream>

using namespace std;

template <typename Type>

class UniquePtr;

template <typename Type>

class SharedPtr;

template <typename Type>

class WeakPtr;

struct Count

{

long countS{ 0 };

long countW{ 0 };

Count(long cS = 1, long cW = 0) noexcept : countS(cS), countW(cW) {}

};

template <typename Type>

class Pointers

{

public:

long use\_count() const noexcept { return rep ? rep->countS : 0; }

Pointers(const Pointers<Type>&) = delete;

Pointers<Type>& operator=(const Pointers<Type>&) = delete;

protected:

Pointers() = default;

Type\* get() const noexcept { return ptr; }

void set(Type\* p, Count\* r) noexcept { ptr = p; rep = r; }

void delShared() noexcept;

void delWeak() noexcept;

void delCount() noexcept;

bool \_compare(const Pointers<Type>& right) const noexcept { return this->get() == right.get(); }

void \_swap(Pointers<Type>& right) noexcept

{

std::swap(ptr, right.ptr);

std::swap(rep, right.rep);

}

void \_copyShared(const Pointers<Type>& right) noexcept;

void \_copyWeak(const Pointers<Type>& right) noexcept;

void \_move(Pointers<Type>& right) noexcept;

private:

Type\* ptr{ nullptr };

Count\* rep{ nullptr };

};

#pragma region Method Pointers

template <typename Type>

void Pointers<Type>::delShared() noexcept

{

if (!ptr) return;

(rep->countS)--;

if (!rep->countS)

{

delete ptr;

ptr = nullptr;

delCount();

}

}

template <typename Type>

void Pointers<Type>::delWeak() noexcept

{

if (rep)

{

(rep->countW)--;

delCount();

}

}

template <typename Type>

void Pointers<Type>::delCount() noexcept

{

if (!rep->countS && !rep->countW)

{

delete rep;

rep = nullptr;

}

}

template <typename Type>

void Pointers<Type>::\_copyShared(const Pointers<Type>& right) noexcept

{

if (right.ptr)

(right.rep->countS)++;

ptr = right.ptr;

rep = right.rep;

}

template <typename Type>

void Pointers<Type>::\_copyWeak(const Pointers<Type>& right) noexcept

{

if (right.rep)

(right.rep->countW)++;

ptr = right.ptr;

rep = right.rep;

}

template <typename Type>

void Pointers<Type>::\_move(Pointers<Type>& right) noexcept

{

ptr = right.ptr;

rep = right.rep;

right.ptr = nullptr;

right.rep = nullptr;

}

#pragma endregion

template <typename Type>

class SharedPtr : public Pointers<Type>

{

public:

SharedPtr() = default;

constexpr SharedPtr(nullptr\_t) noexcept {}

explicit SharedPtr(Type\* p);

SharedPtr(const SharedPtr<Type>& other) noexcept;

explicit SharedPtr(const WeakPtr<Type>& other) noexcept;

SharedPtr(SharedPtr<Type>&& right) noexcept;

SharedPtr(UniquePtr<Type>&& right);

~SharedPtr();

SharedPtr<Type>& operator=(const SharedPtr<Type>& vright) noexcept;

SharedPtr<Type>& operator=(SharedPtr<Type>&& vright) noexcept;

SharedPtr<Type>& operator=(UniquePtr<Type>&& vright);

Type& operator\*() const noexcept { return \*this->get(); }

Type\* operator->() const noexcept { return this->get(); }

explicit operator bool() const noexcept { return this->get() != nullptr; }

bool unique() const noexcept { return this->use\_count() == 1; }

void swap(SharedPtr<Type>& right) noexcept { this->\_swap(right); }

void reset(Type\* p = nullptr) noexcept { (p ? SharedPtr(p) : SharedPtr()).swap(\*this); }

};

# pragma region Methods SharedPtr

template <typename Type>

SharedPtr<Type>::SharedPtr(Type\* p)

{

this->set(p, new Count());

}

template <typename Type>

SharedPtr<Type>::SharedPtr(const SharedPtr<Type>& other) noexcept

{

this->\_copyShared(other);

}

template <typename Type>

SharedPtr<Type>::SharedPtr(const WeakPtr<Type>& other) noexcept

{

this->\_copyShared(other);

}

template <typename Type>

SharedPtr<Type>::SharedPtr(SharedPtr<Type>&& right) noexcept

{

this->\_move(right);

}

template <typename Type>

SharedPtr<Type>::SharedPtr(UniquePtr<Type>&& vright)

{

Type\* p = vright.release();

if (p)

this->set(p, new Count());

}

template <typename Type>

SharedPtr<Type>::~SharedPtr()

{

this->delShared();

}

template <typename Type>

SharedPtr<Type>& SharedPtr<Type>::operator=(const SharedPtr<Type>& vright) noexcept

{

if (this->\_compare(vright)) return \*this;

this->delShared();

this->\_copyShared(vright);

return \*this;

}

template <typename Type>

SharedPtr<Type>& SharedPtr<Type>::operator=(SharedPtr<Type>&& vright) noexcept

{

if (this->\_compare(vright)) return \*this;

this->delShared();

this->\_move(vright);

return \*this;

}

template <typename Type>

SharedPtr<Type>& SharedPtr<Type>::operator=(UniquePtr<Type>&& vright)

{

this->delShared();

Type\* p = vright.release();

this->set(p, p ? new Count() : nullptr);

return \*this;

}

#pragma endregion

template <typename Type>

class WeakPtr : public Pointers<Type>

{

public:

WeakPtr() = default;

WeakPtr(const WeakPtr<Type>& other) noexcept;

WeakPtr(const SharedPtr<Type>& other) noexcept;

WeakPtr(WeakPtr<Type>&& other) noexcept;

~WeakPtr();

WeakPtr<Type>& operator=(const WeakPtr<Type>& vright) noexcept;

WeakPtr<Type>& operator=(const SharedPtr<Type>& vright) noexcept;

WeakPtr<Type>& operator=(WeakPtr<Type>&& vright) noexcept;

void reset() noexcept { WeakPtr().swap(\*this); }

void swap(WeakPtr<Type>& other) noexcept { this->\_swap(other); }

bool expired() const noexcept { return this->use\_count() == 0; }

SharedPtr<Type> lock()const noexcept { return SharedPtr<Type>(\*this); }

};

# pragma region Methods WeakPtr

template <typename Type>

WeakPtr<Type>::WeakPtr(const WeakPtr<Type>& other) noexcept

{

this->\_copyWeak(other);

}

template <typename Type>

WeakPtr<Type>::WeakPtr(const SharedPtr<Type>& other) noexcept

{

this->\_copyWeak(other);

}

template <typename Type>

WeakPtr<Type>::WeakPtr(WeakPtr<Type>&& other) noexcept

{

this->\_move(other);

}

template <typename Type>

WeakPtr<Type>::~WeakPtr()

{

this->delWeak();

}

template <typename Type>

WeakPtr<Type>& WeakPtr<Type>::operator=(const WeakPtr<Type>& vright) noexcept

{

if (this->\_compare(vright)) return \*this;

this->delWeak();

this->\_copyWeak(vright);

return \*this;

}

template <typename Type>

WeakPtr<Type>& WeakPtr<Type>::operator=(const SharedPtr<Type>& vright) noexcept

{

if (this->\_compare(vright)) return \*this;

this->delWeak();

this->\_copyWeak(vright);

return \*this;

}

template <typename Type>

WeakPtr<Type>& WeakPtr<Type>::operator=(WeakPtr<Type>&& vright) noexcept

{

if (this->\_compare(vright)) return \*this;

this->delWeak();

this->\_move(vright);

return \*this;

}

#pragma endregion

class A

{

int key;

public:

A(int k) : key(k)

{

cout << "Calling the constructor of class A (obj" << key << ");" << endl;

}

~A()

{

cout << "Calling a class A destructor (obj" << key << ");" << endl;

}

void f() { cout << "Method f;" << endl; }

};

void main()

{

SharedPtr<A> obj1(new A(1));

obj1->f();

SharedPtr<A> s1, s2(obj1), s3;

s2->f();

cout << s2.use\_count() << endl;

WeakPtr<A> w1 = s2;

s1 = w1.lock();

SharedPtr<A> s4(w1);

cout << s2.use\_count() << endl;

WeakPtr<A> w2;

{

SharedPtr<A> obj2(new A(2));

w2 = obj2;

if (!w2.expired())

(w2.lock())->f();

}

if (!w2.expired())

(w2.lock())->f();

s2.reset();

s3 = s1;

}

Пример 10.04. Создание итератора (без проверок и обработки исключительных ситуация).

# include <iostream>

# include <memory>

# include <iterator>

# include <initializer\_list>

using namespace std;

template <typename Type>

class Iterator;

class BaseArray

{

public:

BaseArray(size\_t sz = 0) { count = shared\_ptr<size\_t>(new size\_t(sz)); }

virtual ~BaseArray() = default;

size\_t size() { return bool(count) ? \*count : 0; }

operator bool() { return size(); }

protected:

shared\_ptr<size\_t> count;

};

template <typename Type>

class Array final : public BaseArray

{

public:

Array(initializer\_list<Type> lt);

virtual ~Array() {}

Iterator<Type> begin() const { return Iterator<Type>(arr, count); }

Iterator<Type> end() const { return Iterator<Type>(arr, count, \*count); }

private:

shared\_ptr<Type[]> arr{ nullptr };

};

template <typename Type>

class Iterator : public std::iterator<std::input\_iterator\_tag, Type>

{

friend class Array<Type>;

private:

Iterator(const shared\_ptr<Type[]>& a, const shared\_ptr<size\_t>& c, size\_t ind = 0) : arr(a), count(c), index(ind) {}

public:

Iterator(const Iterator& it) = default;

bool operator!=(Iterator const& other) const;

bool operator==(Iterator const& other) const;

Type& operator\*();

const Type& operator\*() const;

Type\* operator->();

const Type\* operator->() const;

Iterator<Type>& operator++();

Iterator<Type> operator++(int);

private:

weak\_ptr<Type[]> arr;

weak\_ptr<size\_t> count;

size\_t index = 0;

};

#pragma region Method Array

template <typename Type>

Array<Type>::Array(initializer\_list<Type> lt)

{

if (!(\*count = lt.size())) return;

arr = shared\_ptr<Type[]>(new Type[\*count]);

size\_t i = 0;

for (Type elem : lt)

arr[i++] = elem;

}

#pragma endregion

#pragma region Methods Iterator

template <typename Type>

bool Iterator<Type>::operator!=(Iterator const& other) const { return index != other.index; }

template <typename Type>

Type& Iterator<Type>::operator\*()

{

shared\_ptr<Type[]> a(arr);

return a[index];

}

template <typename Type>

Iterator<Type>& Iterator<Type>::operator++()

{

shared\_ptr<size\_t> n(count);

if (index < \*n)

index++;

return \*this;

}

template <typename Type>

Iterator<Type> Iterator<Type>::operator++(int)

{

Iterator<Type> it(\*this);

++(\*this);

return it;

}

#pragma endregion

template <typename Type>

ostream& operator<<(ostream& os, const Array<Type>& arr)

{

for (auto elem : arr)

cout << elem << " ";

return os;

}

int main()

{

Array<int> arr{ 1, 2, 3, 4, 5 };

cout << " Array: " << arr << endl;

}