Пример 10.01. Использование итераторов для массива С и контенеров.

# include <iostream>

# include <vector>

# include <list>

# include <iterator>

# include <concepts>

using namespace std;

template <input\_iterator Iter>

void print(Iter&& first, Iter&& last)

{

for (auto it = first; it != last; ++it)

cout << \*it << ' ';

cout << endl;

}

int main()

{

int v1[]{ 1, 2, 3, 4, 5 };

cout << "iterator array: ";

print(begin(v1), end(v1));

vector v2{ 1, 2, 3, 4, 5 };

cout << "iterator vector: ";

print(v2.begin(), v2.end());

cout << "const iterator vector: ";

print(v2.cbegin(), v2.cend());

cout << "reverse\_iterator vector: ";

print(v2.rbegin(), v2.rend());

cout << "const\_reverse\_iterator vector: ";

print(v2.crbegin(), v2.crend());

const vector v3{ 1, 2, 3, 4, 5 };

cout << "const\_iterator vector: ";

print(v3.begin(), v3.end());

cout << "const\_reverse\_iterator: ";

print(v3.rbegin(), v3.rend());

list l{ 1, 2, 3, 4, 5 };

cout << "iterator list: ";

print(l.begin(), l.end());

}

Пример 10.02. Использование оператора -> для итераторов.

# include <iostream>

# include <vector>

# include <iterator>

using namespace std;

class A

{

private:

int a;

static int q;

public:

A() { a = ++q; }

void f() { cout << a << endl; }

};

int A::q = 0;

int main()

{

vector<A> vec(10);

for (auto it = vec.begin(); it != vec.end(); ++it)

it->f();

}

Пример 10.03. Пример вложенного итератора.

# include <iostream>

# include <iterator>

using namespace std;

/\*

template <

typename Category, // категория итератора

typename T, // тип значения

typename Distance = ptrdiff\_t, // тип расстояния между итераторами

typename Pointer = T\*, // указатель на значение

typename Reference = T& // ссылка на значение

> struct iterator;

\*/

template <int FROM, int TO>

class Range

{

public:

class iterator

{

private:

int num = FROM;

public:

using iterator\_category = input\_iterator\_tag;

using value\_type = int;

using difference\_type = int;

using pointer = const int\*;

using reference = const int&;

explicit iterator(long nm = 0) : num(nm) {}

iterator& operator ++() { num += FROM <= TO ? 1 : -1; return \*this; }

iterator operator ++(int) { iterator retval = \*this; ++(\*this); return retval; }

bool operator ==(iterator other) const { return num == other.num; }

bool operator !=(iterator other) const { return !(\*this == other); }

reference operator\*() const { return num; }

};

iterator begin() { return iterator(FROM); }

iterator end() { return iterator(FROM <= TO ? TO + 1 : TO - 1); }

};

int main()

{

auto rng = Range<15, 25>();

cout << "count elem = " << distance(rng.begin(), rng.end()) << endl;

for (auto it = find(rng.begin(), rng.end(), 20); it != rng.end(); ++it)

{

cout << \*it << ' ';

}

cout << endl;

for (auto i : Range<5, 2>())

{

cout << i << ' ';

}

cout << endl;

}

Пример 10.04. Пример: числа Фибоначчи.

# include <iostream>

using namespace std;

struct fibonacci { int num{ 0 }; };

class Fibiter

{

private:

int cur{ 1 }, prv{ 0 };

public:

Fibiter() = default;

Fibiter& operator ++()

{

prv = exchange(cur, cur + prv);

return \*this;

}

int operator \*() { return cur; }

auto operator <=>(const Fibiter&) const = default;

};

Fibiter begin(fibonacci) { return Fibiter{}; }

Fibiter end(fibonacci fib)

{

Fibiter it;

while (\*it <= fib.num) ++it;

return it;

}

int main()

{

for (auto el : fibonacci{ 100 })

cout << el << ' ';

cout << endl;

for (auto it = begin(fibonacci{}); it != end(fibonacci{ 1000 }); ++it)

cout << \*it << ' ';

cout << endl;

}

Пример 10.05. Реализация copy.

# include <iostream>

# include <concepts>

# include <list>

# include <vector>

# include <iterator>

using namespace std;

namespace my

{

template <input\_iterator InputIt,

output\_iterator<typename iterator\_traits<InputIt>::value\_type> OutputIt>

auto copy(InputIt first, InputIt last, OutputIt dfirst)

{

for (auto it = first; it != last; ++it, ++dfirst)

\*dfirst = \*it;

return dfirst;

}

}

int main()

{

list l{ 1, 2, 3, 4, 5 };

vector<int> v;

my::copy(l.begin(), l.end(), std::back\_inserter(v));

my::copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

}

Пример 10.06. Концепты итераторов и реализация distance, advance с их использованием.

# include <iostream>

# include <iterator>

# include <vector>

# include <list>

using namespace std;

template <typename I>

concept Iterator = requires()

{

typename I::value\_type;

typename I::difference\_type;

typename I::pointer;

typename I::reference;

};

template <typename T, typename U>

concept DerivedFrom = is\_base\_of<U, T>::value;

# pragma region Input\_Iterator

template<typename T>

concept EqualityComparable = requires(T a, T b)

{

{ a == b } -> same\_as<bool>;

{ a != b } -> same\_as<bool>;

};

template <typename I>

concept InputIterator = Iterator<I> &&

requires { typename I::iterator\_category; }&&

EqualityComparable<I>&&

DerivedFrom<typename I::iterator\_category, input\_iterator\_tag>;

# pragma endregion

# pragma region Forward\_Iterator

template <typename I>

concept Incrementable = requires(I it)

{

{ ++it } -> same\_as<I&>;

{ it++ } -> same\_as<I>;

};

template <typename I>

concept ForwardIterator = InputIterator<I> &&

Incrementable<I> &&

DerivedFrom<typename I::iterator\_category, forward\_iterator\_tag>;

# pragma endregion

# pragma region Bidirectional\_Iterator

template <typename I>

concept Decrementable = requires(I it)

{

{ --it } -> same\_as<I&>;

{ it-- } -> same\_as<I>;

};

template <typename I>

concept BidirectionalIterator = ForwardIterator<I> &&

Decrementable<I> &&

DerivedFrom<typename I::iterator\_category, bidirectional\_iterator\_tag>;

# pragma endregion

# pragma region Random\_Access\_Iterator

template <typename I>

concept RandomAccess = requires(I it, typename I::difference\_type n)

{

{ it + n } -> same\_as<I>;

{ it - n } -> same\_as<I>;

{ it += n } -> same\_as<I&>;

{ it -= n } -> same\_as<I&>;

{ it[n] } -> same\_as<typename I::reference>;

};

template <typename I>

concept Distance = requires(I it1, I it2)

{

{ it2 - it1 } -> convertible\_to<typename I::difference\_type>;

};

template <typename I>

concept RandomAccessIterator = BidirectionalIterator<I> &&

RandomAccess<I> && Distance<I> &&

DerivedFrom<typename I::iterator\_category, random\_access\_iterator\_tag>;

# pragma endregion

namespace my

{

# define V\_4

# ifdef V\_1

template <InputIterator Iter>

typename Iter::difference\_type distance(Iter first, Iter last)

{

typename Iter::difference\_type count = 0;

for (Iter current = first; current != last; ++current, ++count);

return count;

}

template <RandomAccessIterator Iter>

typename Iter::difference\_type distance(Iter first, Iter last)

{

return last - first;

}

# elif defined(V\_2)

template <InputIterator Iter>

auto distance(Iter first, Iter last)

{

typename Iter::difference\_type count = 0;

for (Iter current = first; current != last; ++current, ++count);

return count;

}

template <RandomAccessIterator Iter>

auto distance(Iter first, Iter last)

{

return last - first;

}

# elif defined(V\_3)

template<InputIterator Iter>

constexpr auto distance(Iter first, Iter last)

{

if constexpr (RandomAccessIterator<Iter>)

{

return last - first;

}

else

{

iter\_difference\_t<Iter> count{};

for (auto current = first; current != last; ++current, ++count);

return count;

}

}

# elif defined(V\_4)

constexpr auto distance(InputIterator auto first, InputIterator auto last)

{

if constexpr (is\_same\_v<decltype(first), decltype(last)>)

{

iter\_difference\_t<decltype(first)> count{};

for (auto current = first; current != last; ++current, ++count);

return count;

}

}

constexpr auto distance(RandomAccessIterator auto first, RandomAccessIterator auto last)

{

if constexpr (is\_same\_v<decltype(first), decltype(last)>)

{

return last - first;

}

}

# endif

template <InputIterator Iter, typename Dist>

void advance(Iter& it, Dist n)

{

for (auto dist = typename Iter::difference\_type(n); dist > 0; --dist, ++it);

}

template <BidirectionalIterator Iter, typename Dist>

void advance(Iter& it, Dist n)

{

auto dist = typename Iter::difference\_type(n);

typename Iter::difference\_type step{ dist > 0 ? 1 : -1 };

for (; step \* dist > 0; (dist > 0 ? ++it : --it), dist -= step);

}

template <RandomAccessIterator Iter, typename Dist>

void advance(Iter& it, Dist n)

{

auto dist = typename Iter::difference\_type(n);

it += dist;

}

}

int main()

{

vector<double> v(100);

auto iv = v.begin();

my::advance(iv, 3);

cout << my::distance(iv, v.end()) << endl;

list<double> l(10);

auto il = l.begin();

my::advance(il, 3);

cout << my::distance(il, l.end()) << endl;

}

Пример 10.07. Пример итератора (без проверок и обработки исключительных ситуация).

# include <iostream>

# include <memory>

# include <iterator>

# include <initializer\_list>

using namespace std;

template <typename Type>

class Iterator;

template <typename Type>

class ConstIterator;

class BaseArray

{

public:

using size\_type = size\_t;

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

virtual ~BaseArray() = 0;

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

explicit operator bool() const noexcept { return size(); }

protected:

shared\_ptr<size\_t> count;

};

BaseArray::~BaseArray() = default;

template <typename Type>

class Array final : public BaseArray

{

public:

using value\_type = Type;

using iterator = Iterator<Type>;

using const\_iterator = ConstIterator<Type>;

Array(initializer\_list<Type> lt);

~Array() override = default;

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

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

private:

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

};

template <typename Type>

class Iterator

{

friend class Array<Type>;

public:

using iterator\_category = forward\_iterator\_tag;

using value\_type = Type;

using difference\_type = ptrdiff\_t;

using pointer = Type\*;

using reference = Type&;

public:

Iterator(const Iterator& it) = default;

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

reference operator\*();

const reference operator\*() const;

pointer operator->();

const pointer operator->() const;

operator bool() const;

Iterator& operator++();

Iterator operator++(int);

private:

Iterator(const shared\_ptr<Type[]>& a, const shared\_ptr<size\_t>& c, size\_t ind = 0)

: arr(a), count(c), index(ind) {}

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) : BaseArray(lt.size())

{

if (!count) return;

arr = make\_shared<Type[]>(\*count);

for (size\_t i = 0; auto 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>

Iterator<Type>::reference 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>

concept Container = requires(Type t)

{

typename Type::value\_type;

typename Type::size\_type;

typename Type::iterator;

typename Type::const\_iterator;

{ t.size() } noexcept -> same\_as<typename Type::size\_type>;

{ t.end() } noexcept -> same\_as<typename Type::iterator>;

{ t.begin() } noexcept -> same\_as<typename Type::iterator>;

};

ostream& operator <<(ostream & os, const Container auto& container)

{

for (auto elem : container)

cout << elem << " ";

return os;

}

int main()

{

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

cout << "Count = " << distance(arr.begin(), arr.end()) << endl;

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

}

Пример 10.08. Диапазон из итераторов.

# include <iostream>

# include <vector>

using namespace std;

template <input\_iterator Iter>

class Range

{

public:

using value\_type = Iter::value\_type;

using size\_type = size\_t;

using iterator = Iter;

using const\_iterator = const Iter;

private:

Iter first, last;

public:

Range(Iter fst, Iter lst) : first(fst), last(lst) {}

size\_t size() const noexcept;

iterator begin() { return first; }

iterator end() { return last; }

};

template <input\_iterator Iter>

size\_t Range<Iter>::size() const noexcept

{

return distance(first, last);

}

int main()

{

vector v{ 1, 2, 3, 4, 5 };

Range range{ v.begin(), v.end() };

cout << "count = " << range.size() << "; elems: ";

for (auto elem : range)

cout << elem << " ";

cout << endl;

}

Пример 10.09. Реализация zip и zip итератора.

# include <iostream>

# include <vector>

# include <list>

using namespace std;

template <typename Type>

concept Container = requires(Type t)

{

typename Type::value\_type;

typename Type::size\_type;

typename Type::iterator;

typename Type::const\_iterator;

{ t.size() } noexcept -> same\_as<typename Type::size\_type>;

{ t.end() } noexcept -> same\_as<typename Type::iterator>;

{ t.begin() } noexcept -> same\_as<typename Type::iterator>;

};

template <input\_iterator KIter, input\_iterator VIter>

class ZipIterator

{

private:

using keys\_type = typename iterator\_traits<KIter>::value\_type;

using values\_type = typename iterator\_traits<VIter>::value\_type;

using keys\_reference = typename iterator\_traits<KIter>::reference;

using values\_reference = typename iterator\_traits<VIter>::reference;

template <typename Reference>

struct Proxy

{

Reference r;

Reference\* operator ->() { return &r; }

};

public:

using iterator\_category = forward\_iterator\_tag;

using value\_type = pair<keys\_type, values\_type>;

using difference\_type = ptrdiff\_t;

using reference = pair<keys\_reference, values\_reference>;

using pointer = Proxy<reference>;

private:

KIter kiter;

VIter viter;

public:

ZipIterator(KIter kit, VIter vit) : kiter(kit), viter(vit) {}

ZipIterator& operator ++() { ++kiter; ++viter; return \*this; }

ZipIterator operator ++(int) { ZipIterator temp(kiter, viter); ++kiter; ++viter; return temp; }

pointer operator ->() { return pointer{{\*kiter, \*viter}}; }

reference operator \*() { return {\*kiter, \*viter}; }

bool operator ==(const ZipIterator& other) const

{

return kiter == other.kiter && viter == other.viter;

}

};

template <Container Keys, Container Values>

requires requires(Keys k, Values v) { k.size() == v.size(); }

class Zip

{

private:

using keys\_iterator = typename remove\_reference\_t<Keys>::iterator;

using values\_iterator = typename remove\_reference\_t<Values>::iterator;

using keys\_const\_iterator = typename remove\_reference\_t<Keys>::const\_iterator;

using values\_const\_iterator = typename remove\_reference\_t<Values>::const\_iterator;

public:

using value\_type = pair<typename Keys::value\_type, typename Values::value\_type>;

using size\_type = Keys::size\_type;

using iterator = ZipIterator<keys\_iterator, values\_iterator>;

using const\_iterator = ZipIterator<keys\_const\_iterator, values\_const\_iterator>;

private:

Keys& keys;

Values& values;

public:

Zip(Keys& ks, Values& vs) : keys(ks), values(vs) {}

iterator begin() noexcept { return iterator(keys.begin(), values.begin()); }

iterator end() noexcept { return iterator(keys.end(), values.end()); }

const\_iterator begin() const noexcept

{ return const\_iterator(keys.cbegin(), values.cbegin()); }

const\_iterator end() const noexcept

{ return const\_iterator(keys.cend(), values.cend()); }

size\_type size() const noexcept { return keys.size(); }

};

template <typename First, typename Second>

ostream& operator <<(ostream& os, const pair<First, Second>& pr)

{

return os << "(" << pr.first << ", " << pr.second << ")";

}

ostream& operator <<(ostream& os, const Container auto& container)

{

for (auto&& elem : container)

cout << elem << " ";

return os;

}

int main()

{

vector v{ 1, 2, 3, 4, 5 };

list l{ 7.2, 1.3, 4.4, 8.1, 5.6 };

Zip zip(v, l);

cout << "count = " << distance(zip.begin(), zip.end()) << endl;

cout << "zip: " << zip << endl;

}

Пример 10.10. Приведение типов в С++. Использование static\_cast и dynamic\_cast.

# 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 10.11. dynamic\_cast – приведение между базовыми классами.

# include <iostream>

using namespace std;

class Base

{

public:

virtual ~Base() = default;

virtual void f() = 0;

};

class A : public Base

{

public:

void f() override { cout << "function f (class A)" << endl; }

};

class B

{

public:

virtual ~B() = default;

virtual void g() = 0;

};

class C : public A, public B

{

public:

void f() override { cout << "function f (class C)" << endl; }

void g() override { cout << "function g" << endl; }

};

int main()

{

A\* pa = new C;

pa->f();

// pa->g(); // Error!

B\* pb1 = dynamic\_cast<B\*>(pa);

// pb1->f(); // Error!

pb1->g();

Base\* p = dynamic\_cast<Base\*>(pb1);

p->f();

B\* pb2 = dynamic\_cast<B\*>(p);

pb2->g();

delete pa;

}

Пример 10.12. Использование dynamic\_cast для приведения типа между родительскими классами при множественном наследовании.

# include <iostream>

# include <vector>

# include <memory>

using namespace std;

class AbstractVisitor

{

public:

virtual ~AbstractVisitor() = default;

};

template <typename T>

class Visitor

{

public:

virtual ~Visitor() = default;

virtual void visit(const T&) const = 0;

};

class Shape

{

public:

Shape() = default;

virtual ~Shape() = default;

virtual void accept(const AbstractVisitor&) const = 0;

};

class Circle : public Shape

{

private:

double radius;

public:

Circle(double radius) : radius(radius) {}

void accept(const AbstractVisitor& v) const override

{

auto cv = dynamic\_cast<const Visitor<Circle>\*>(&v);

if (cv)

{

cv->visit(\*this);

}

}

};

class Square : public Shape

{

private:

double side;

public:

Square(double side) : side(side) {}

void accept(const AbstractVisitor& v) const override

{

auto cv = dynamic\_cast<const Visitor<Square>\*>(&v);

if (cv)

{

cv->visit(\*this);

}

}

};

class DrawCircle : public Visitor<Circle>

{

void visit(const Circle& circle) const override

{

cout << "Circle" << endl;

}

};

class DrawSquare : public Visitor<Square>

{

void visit(const Square& circle) const override

{

cout << "Square" << endl;

}

};

class Draw : public AbstractVisitor, public DrawCircle, public DrawSquare { };

class DrawAll

{

public:

void operator ()(const vector<unique\_ptr<Shape>>& shapes)

{

for (const auto& s : shapes)

{

s->accept(Draw{});

}

}

};

int main()

{

using Shapes = vector<unique\_ptr<Shape>>;

Shapes shapes;

shapes.emplace\_back(make\_unique<Circle>(1.));

shapes.emplace\_back(make\_unique<Square>(2.));

DrawAll{}(shapes);

}

Пример 10.13. Использование reinterpret\_cast.

# include <iostream>

# include <string.h>

using namespace std;

class A

{

private:

int a{ 63 };

char s[6];

public:

A(const char\* st) { strcpy\_s(s, st); }

};

void print(const char\* st, size\_t len)

{

for (size\_t i = 0; i < len; i++)

cout << st[i];

}

int main()

{

A obj("Ok!!!");

char\* pByte = reinterpret\_cast<char\*>(&obj);

print(pByte, sizeof(obj));

}

Пример 10.14. Реализация move.

# include <iostream>

using namespace std;

namespace my

{

template <typename T>

struct remove\_reference { using type = T; };

template <typename T>

struct remove\_reference<T&> { using type = T; };

template <typename T>

struct remove\_reference<T&&> { using type = T; };

template <typename T>

using remove\_reference\_t = typename remove\_reference<T>::type;

# define V\_1

# ifdef V\_1

template <typename T>

typename remove\_reference<T>::type&& move(T&& t)

{

return static\_cast<typename remove\_reference<T>::type&&>(t);

}

# elif defined(V\_2)

template <typename T>

remove\_reference\_t<T>&& move(T&& t)

{

return static\_cast<remove\_reference\_t<T>&&>(t);

}

# elif defined(V\_3)

decltype(auto) move(auto&& t)

{

return static\_cast<remove\_reference\_t<decltype(t)>&&>(t);

}

# endif

}

class A

{

public:

A() { cout << "constructor" << endl; }

A(const A& over) { cout << "copy constructor" << endl; }

A(A&& over) noexcept { cout << "move constructor" << endl; }

~A() { cout << "destructor" << endl; }

A& operator =(const A& over)

{

cout << "copy assignment operator" << endl;

return \*this;

}

A& operator =(A&& over) noexcept

{

cout << "move assignment operator" << endl;

return \*this;

}

};

template <typename Type>

void mySwap(Type& d1, Type& d2)

{

Type dt = my::move(d1);

d1 = my::move(d2);

d2 = my::move(dt);

}

int main()

{

A obj1, obj2;

mySwap(obj1, obj2);

}

Пример 10.15. Реализация forward.

# include <iostream>

# include <memory>

using namespace std;

namespace my

{

template <typename T>

struct remove\_reference { using type = T; };

template <typename T>

struct remove\_reference<T&> { using type = T; };

template <typename T>

struct remove\_reference<T&&> { using type = T; };

template <typename T>

using remove\_reference\_t = typename remove\_reference<T>::type;

template <typename T>

constexpr T&& forward(remove\_reference\_t<T>& value) noexcept

{

return static\_cast<T&&>(value);

}

template <typename T>

constexpr T&& forward(remove\_reference\_t<T>&& value) noexcept

{

return static\_cast<T&&>(value);

}

}

template <typename T, typename... Args>

shared\_ptr<T> create(Args&&... args)

{

return shared\_ptr<T>(new T(my::forward<Args>(args)...));

}

struct Person

{

Person(const string& name) { cout << "copy constructor" << endl; }

Person(string&& name) { cout << "move constructor" << endl; }

};

int main()

{

shared\_ptr<Person> p1 = create<Person>("Ok!!!");

string nm("name");

shared\_ptr<Person> p2 = create<Person>(nm);

}

Пример 10.16. Реализация addressof.

# include <iostream>

using namespace std;

class A

{

private:

int a;

public:

A\* operator &() const noexcept = delete;

};

namespace my

{

template<typename T>

T\* addressof(T& v)

{

return reinterpret\_cast<T\*>(&const\_cast<char&>(reinterpret\_cast<const char&>(v)));

}

}

int main()

{

A obj;

// cout << &obj << endl; // Error!

hex(cout);

cout << my::addressof<A>(obj) << endl;

}