Пример 08.01. Шаблоны функций.

# include <iostream>

using namespace std;

template <typename Type>

Type\* initArray(int count);

template <typename Type>

void freeArray(Type\* arr);

template <typename Type>

Type\* inputArray(Type\* arr, int cnt);

template <typename Type>

void outputArray(Type\* arr, int cnt);

template <typename Type> using Tfunc = int (\*)(const Type&, const Type&);

template <typename Type>

void sort(Type\* arr, int cnt, Tfunc<Type> cmp);

int compare(const double& d1, const double& d2)

{

return d1 - d2;

}

void main()

{

const int N = 10;

double\* arr = initArray<double>(N);

cout << "Enter array: ";

inputArray(arr, N);

sort(arr, N, compare);

cout << "Resulting array: ";

outputArray(arr, N);

freeArray(arr);

}

template <typename Type>

Type\* initArray(int count)

{

return new Type[count];

}

template <typename Type>

void freeArray(Type\* arr)

{

delete[] arr;

}

template <typename Type>

Type\* inputArray(Type\* arr, int cnt)

{

for (int i = 0; i < cnt; i++)

cin >> arr[i];

return arr;

}

template <typename Type>

void outputArray(Type\* arr, int cnt)

{

for (int i = 0; i < cnt; i++)

cout << arr[i] << " ";

cout << endl;

}

template <typename Type>

void sort(Type\* arr, int cnt, Tfunc<Type> cmp)

{

for (int i = 0; i < cnt - 1; i++)

for (int j = i + 1; j < cnt; j++)

if (cmp(arr[i], arr[j]) > 0)

std::swap(arr[i], arr[j]);

}

Пример 08.29. Выведение типа и стирание ссылок и const.

# include <iostream>

# define V\_1

# ifdef V\_1

template <typename T>

T f(T v) { return v; }

# elif defined(V\_2)

template <typename T>

T f(T& v) { return v; }

# elif defined(V\_3)

template <typename T>

T f(const T& v) { return v; }

# elif defined(V\_4)

template <typename T>

T f(T&& v) { return v; }

# elif defined(V\_5)

template <typename T>

T& f(T&& v) { return v; }

# elif defined(V\_6)

template <typename T>

T&& f(T&& v) { return std::forward<T>(v); }

# elif defined(V\_7)

auto f(auto v) { return v; }

# elif defined(V\_8)

auto f(auto& v) { return v; }

# elif defined(V\_9)

auto f(const auto& v) { return v; }

# elif defined(V\_10)

auto&& f(auto&& v) { return v; }

# elif defined(V\_11)

auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V\_12)

decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V\_13)

template <typename T>

auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

# endif

int main()

{

int i;

int& a = i;

const int& b = 0;

decltype(auto) r1 = f(i);

// 1. T f(T v) ---> int f<int>(int)

// 2. T f(T& v) ---> int f<int>(int&)

// 3. T f(const T& v) ---> int f<int>(const int&)

// 4. T f(T&& v) ---> int& f<int&>(int&)

// 5. T& f(T&& v) ---> int& f<int&>(int&)

// 6. T&& f(T&& v) { return std::forward<T>(v); } ---> int& f<int&>(int&)

// 7. auto f(auto v) ---> int f<int>(int)

// 8. auto f(auto& v) ---> int f<int>(int&)

// 9. auto f(const auto& v) ---> int f<int>(const int&)

// 10. auto&& f(auto&& v) ---> int& f<int&>(int&)

// 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int& f<int&>(int&)

// 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int& f<int&>(int&)

// 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

// ---> int& f<int&>(int&)

decltype(auto) r2 = f(a);

// 1. T f(T v) ---> int f<int>(int)

// 2. T f(T& v) ---> int f<int>(int&)

// 3. T f(const T& v) ---> int f<int>(const int&)

// 4. T f(T&& v) ---> int& f<int&>(int&)

// 5. T& f(T&& v) ---> int& f<int&>(int&)

// 6. T&& f(T&& v) { return std::forward<T>(v); } ---> int& f<int&>(int&)

// 7. auto f(auto v) ---> int f<int>(int)

// 8. auto f(auto& v) ---> int f<int>(int&)

// 9. auto f(const auto& v) ---> int f<int>(const int&)

// 10. auto&& f(auto&& v) ---> int& f<int&>(int&)

// 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int& f<int&>(int&)

// 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int& f<int&>(int&)

// 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

// ---> int& f<int&>(int&)

decltype(auto) r3 = f(b);

// 1. T f(T v) ---> int f<int>(int)

// 2. T f(T& v) ---> const int f<const int>(const int&)

// 3. T f(const T& v) ---> int f<int>(const int&)

// 4. T f(T&& v) ---> const int& f<const int&>(const int&)

// 5. T& f(T&& v) ---> const int& f<const int&>(const int&)

// 6. T&& f(T&& v) { return std::forward<T>(v); }

// ---> const int& f<const int&>(const int&)

// 7. auto f(auto v) ---> int f<int>(int)

// 8. auto f(auto& v) ---> int f<const int>(const int&)

// 9. auto f(const auto& v) ---> int f<int>(const int&)

// 10. auto&& f(auto&& v) ---> const int& f<const int&>(const int&)

// 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> const int& f<const int&>(const int&)

// 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> const int& f<const int&>(const int&)

// 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

// ---> const int& f<const int&>(const int&)

decltype(auto) r4 = f(std::move(a));

// 1. T f(T v) ---> int f<int>(int)

// 2. T f(T& v) ---> Error!

// 3. T f(const T& v) ---> int f<int>(const int&)

// 4. T f(T&& v) ---> int f<int>(int)

// 5. T& f(T&& v) ---> int& f<int>(int&&)

// 6. T&& f(T&& v) { return std::forward<T>(v); } ---> int&& f<int>(int&&)

// 7. auto f(auto v) ---> int f<int>(int)

// 8. auto f(auto& v) ---> Error!

// 9. auto f(const auto& v) ---> int f<int>(const int&)

// 10. auto&& f(auto&& v) ---> int f<int>(int&&)

// 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int&& f<int>(int&&)

// 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

// ---> int&& f<int>(int&&)

// 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

// ---> int&& f<int>(int&&)

}

Пример 08.02. Правило вызова функций и шаблонов функций.

# include <iostream>

template <typename Type>

void swap(Type& val1, Type& val2)

{

Type temp = std::move(val1);

val1 = std::move(val2);

val2 = std::move(temp);

}

template<>

void swap<float>(float& val1, float& val2)

{

float temp = val1;

val1 = val2;

val2 = temp;

}

void swap(float& val1, float& val2)

{

float temp = val1;

val1 = val2;

val2 = temp;

}

void swap(int& val1, int& val2)

{

int temp = val1;

val1 = val2;

val2 = temp;

}

class A

{

public:

A() = default;

A(A&&) noexcept { std::cout << "Move constuctor!" << std::endl; }

A& operator =(A&&) noexcept

{

std::cout << "Move assignment operator!" << std::endl;

return \*this;

}

};

void main()

{

const int N = 2;

int a1[N];

float a2[N];

double a3[N];

A a4[N]{};

swap(a1[0], a1[1]); // swap(int&, int&)

swap<int>(a1[0], a1[1]); // swap<int>(int&, int&)

swap(a2[0], a2[1]); // swap(float&, float&)

swap<float>(a2[0], a2[1]); // swap<>(float&, float&)

swap(a3[0], a3[1]); // swap<double>(double&, double&)

swap(a4[0], a4[1]); // swap<A>(A&, A&)

}

Пример 08.03. “Срезание” ссылок и константности при выводе типа шаблона.

# include <iostream>

using namespace std;

# define PRIM\_05

# ifdef PRIM\_01

template <typename T>

T sum(T x, T y)

{

return x + y;

}

# elif defined(PRIM\_02)

template <typename T>

T sum(T& x, T& y)

{

return x + y;

}

# elif defined(PRIM\_03)

auto sum(auto x, auto y)

{

return x + y;

}

# elif defined(PRIM\_04)

auto sum(auto& x, auto& y)

{

return x + y;

}

# elif defined(PRIM\_05)

auto sum(auto&& x, auto&& y)

{

return x + y;

}

#endif

int main()

{

const int& a = 1, & b = 2;

cout << sum(a, b) << endl;

// 1) T(T, T) -> int sum<int>(int, int)

// 2) T(T&, T&) -> const int sum<const int>(const int&, const int&)

// 3) auto(auto, auto) -> int sum<int, int>(int, int)

// 4) auto(auto&, auto&) -> const int sum<const int, const int>(const int&, const int&)

// 5) auto(auto&&, auto&&) -> const int sum<const int&, const int&>(const int&, const int&)

int c = 3, & d = c;

cout << sum(c, d) << endl;

// 1) T(T, T) -> int sum<int>(int, int)

// 2) T(T&, T&) -> int sum<int>(int&, int&)

// 3) auto(auto, auto) -> int sum<int, int>(int, int)

// 4) auto(auto&, auto&) -> int sum<int, int>(int&, int&)

// 5) auto(auto&&, auto&&) -> int sum<int&, int&>(int&, int&)

}

Пример 08.04. Свертка ссылок и вывод типа шаблонного параметра.

# include <iostream>

using namespace std;

template <typename T>

T f(T&& x) { return x; }

int main()

{

int a;

const int b = 0;

f(a); // int& f<int&>(int&)

f(b); // const int& f<const int&>(const int&)

f(0); // int f<int>(int&&)

}

Пример 08.05. Определение типа возвращаемого значения для шаблона функции.

# include <iostream>

using namespace std;

template <typename T, typename U>

auto sum(const T& elem1, const U& elem2) // -> decltype(elem1 + elem2)

{

return elem1 + elem2;

}

void main()

{

auto s = sum(1, 1.2);

cout << "Result: " << s << endl;

}

Пример 08.06. Специализация шаблона функции.

# include <iostream>

using namespace std;

template <typename T>

T get\_value();

template<>

int get\_value() { return 413; }

template<>

double get\_value() { return 3.14; }

int main()

{

auto x = get\_value<int>();

auto y = get\_value<double>();

// auto z = get\_value<float>(); // Error linker!

cout << " x= " << x << " y = " << y << endl;

}

Пример 08.07. Шаблон класса, шаблоны методов (без обработки исключительных ситуаций).

# include <iostream>

using namespace std;

template <typename Type, size\_t N>

class Array

{

private:

Type arr[N];

public:

Array() = default;

Array(initializer\_list<Type> lt);

Type& operator[](int ind);

const Type& operator[](int ind) const;

bool operator ==(const Array& a) const;

template <typename Type, size\_t N>

friend Array<Type, N> operator+(const Array<Type, N>& a1, const Array<Type, N>& a2);

};

template <typename Type, size\_t N>

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

{

int n = N <= lt.size() ? N : lt.size();

auto it = lt.begin();

int i;

for (i = 0; i < n; i++, ++it)

arr[i] = \*it;

for (; i < N; i++)

arr[i] = 0.;

}

template <typename Type, size\_t N>

Type& Array<Type, N>::operator[](int ind) { return arr[ind]; }

template <typename Type, size\_t N>

const Type& Array<Type, N>::operator[](int ind) const { return arr[ind]; }

template <typename Type, size\_t N>

bool Array<Type, N>::operator ==(const Array& a) const

{

if (this == &a) return true;

bool Key = true;

for (int i = 0; Key && i < N; i++)

Key = arr[i] == a.arr[i];

return Key;

}

template <typename Type, size\_t N>

Array<Type, N> operator +(const Array<Type, N>& a1, const Array<Type, N>& a2)

{

Array<Type, N> res;

for (int i = 0; i < N; i++)

res.arr[i] = a1.arr[i] + a2.arr[i];

return res;

}

template <typename Type, size\_t N>

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

{

for (int i = 0; i < N; i++)

os << a[i] << " ";

return os;

}

int main()

{

Array<double, 3> a1{ 1, 2, 3 }, a2{ 1, 2, 3 }, a3{ 4, 2 };

if (a1 == a2)

a1 = a2 + a3;

cout << a1 << endl;

}

Пример 08.08. Конструктор для вывода типа параметра шаблона класса.

# include <iostream>

using namespace std;

template <typename Type>

class Complex

{

private:

Type re, im;

public:

Complex(Type r, Type i) : re(r), im(i) {}

Type getReal() const { return re; }

Type getImage() const { return im; }

};

template <typename Type>

ostream& operator <<(ostream& os, const Complex<Type>& c)

{

return os << "(" << c.getReal() << "; " << c.getImage() << ")";

}

int main()

{

Complex c(1., 2.);

cout << c << endl;

}

Пример 08.09. Конструкторы для вывода типа параметра шаблона класса.

# include <iostream>

# include <initializer\_list>

# include <vector>

using namespace std;

template <typename Type>

class Array

{

private:

Type\* arr;

size\_t count;

public:

Array(initializer\_list<Type> lst) : count(lst.size())

{

arr = new Type[count]{};

for (size\_t i = 0; auto && elem : lst)

arr[i++] = elem;

}

template <typename Iter>

Array(Iter ib, Iter ie) : count(ie - ib)

{

arr = new Type[count]{};

size\_t i = 0;

for (auto it = ib; it != ie; ++it, i++)

arr[i] = \*it;

}

template <typename U>

friend ostream& operator <<(ostream& os, const Array<U>& ar);

};

// Вывод параметра шаблона класса Array

template <typename Iter>

Array(Iter ib, Iter ie)->Array<typename iterator\_traits<Iter>::value\_type>;

template <typename U>

ostream& operator <<(ostream& os, const Array<U>& ar)

{

if (!ar.count) return os;

os << ar.arr[0];

for (size\_t i = 1; i < ar.count; ++i)

os << ", " << ar.arr[i];

return os;

}

int main()

{

Array a1{ 1., 2., 3. };

cout << a1 << endl;

vector v{ 4., 5., 6. };

auto a2 = Array(v.begin(), v.end());

cout << a2 << endl;

}

Пример 08.10. Шаблонный метод класса.

# include <iostream>

using namespace std;

class A

{

public:

template <typename Type>

const Type& f(const Type& elem);

};

template <typename Type>

const Type& A::f(const Type& elem) { return elem; }

int main()

{

A obj;

cout << obj.f(2.) << endl;

cout << obj.f("String") << endl;

}

Пример 08.11. Шаблонный метод шаблонного класса.

# include <iostream>

using namespace std;

template <typename T>

class A

{

private:

T elem;

public:

A(const T& d) : elem(d) {}

template<typename U>

auto sum(U d); // -> decltype(d + this->A<T>::elem);

};

template<typename T>

template<typename U>

auto A<T>::sum(U d) // -> decltype(d + this->A<T>::elem)

{

return elem + d;

}

int main()

{

A obj(1);

cout << obj.sum(1.1) << endl;

}

Пример 08.12. Свертка ссылок и вывод типа параметра шаблонного метода класса.

# include <iostream>

template <typename T>

class A

{

public:

T f(T&& t) { return t; }

template <typename U>

T g(U&& u) { return u; }

};

int main()

{

A<int> obj{};

int i;

// obj.f(i); // Error!

obj.g(i); // int A<int>::g<int&>(int&)

obj.f(0); // int A<int>::f(int&&)

obj.g(0); // int A<int>::g<int>(int&&)

}

Пример 08.13. Использование decltype на примере шаблонного класса Complex.

# include <iostream>

using namespace std;

template <typename T>

class Complex

{

private:

T real;

T imag;

public:

Complex(const T& r, const T& i) : real(r), imag(i) {}

template <typename U>

auto operator +(const Complex<U>& d) const;

const T& getReal() const { return real; }

const T& getImag() const { return imag; }

};

template <typename T>

template <typename U>

auto Complex<T>::operator +(const Complex<U>& d) const

{

return Complex<decltype(real + d.getReal())>(real + d.getReal(), imag + d.getImag());

}

template <typename Type>

ostream& operator <<(ostream& os, const Complex<Type>& com)

{

return os << "(" << com.getReal() << ", " << com.getImag() << ")" << endl;

}

int main()

{

Complex c1(1.1, 2.2);

Complex c2(1, 2);

cout << c2 + c1 << endl;

}

Пример 08.14. Использование forward для идеальной передачи (lvalue-copy, rvalue-move).

# include <iostream>

using namespace std;

class A

{

public:

A() = default;

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

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

};

template <typename Func, typename Arg>

decltype(auto) call(Func&& func, Arg&& arg)

{

// return func(arg);

return forward<Func>(func)(forward<Arg>(arg));

}

A f(A a) { cout << "f called" << endl; return a; }

int main()

{

A obj{};

auto r1 = call(f, obj);

cout << endl;

auto r2 = call(f, move(obj));

}

Пример 08.15. Параметры шаблона класса по умолчанию.

# include <iostream>

# include <exception>

using namespace std;

template <typename T>

class Default\_delete

{

public:

void operator ()(T\* ptr) { delete ptr; }

};

template <typename Type, typename Deleter = Default\_delete<Type>>

class Holder

{

private:

Type\* ptr;

Deleter del;

public:

Holder(Type\* p = nullptr, Deleter d = Deleter{}) noexcept : ptr(p), del(d) {}

~Holder() { del(ptr); }

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

};

class File\_close

{

public:

void operator ()(FILE\* stream) { fclose(stream); }

};

Holder<FILE, File\_close> make\_file(const char\* filename, const char\* mode)

{

FILE\* stream = fopen(filename, mode);

if (!stream) throw std::runtime\_error("file opening error");

return { stream };

}

auto main() -> int

{

try

{

Holder<FILE, File\_close> stream = make\_file("test.txt", "w");

fputs("Ok!!!", stream.get());

}

catch (const runtime\_error& e)

{

cout << e.what() << endl;

}

}

Пример 08.16. Полная специализация шаблона класса и метода шаблона класса.

# include <iostream>

using namespace std;

template <typename Type>

class A

{

public:

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

void f() { cout << "metod f of template A;" << endl; }

};

template <>

void A<int>::f() { cout << "specialization of metod f of template A;" << endl; }

template <>

class A<float>

{

public:

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

void f() { cout << "metod f specialization template A;" << endl; }

void g() { cout << "metod g specialization template A;" << endl; }

};

int main()

{

A<double> obj1;

obj1.f();

A<float> obj2;

obj2.f();

obj2.g();

A<int> obj3;

obj3.f();

}

Пример 08.17. Частичная специализация шаблона класса, параметры шаблона класса по умолчанию.

# include <iostream>

using namespace std;

template <typename T1, typename T2 = double>

class A

{

public:

A() { cout << "constructor of template A<T1, T2>;" << endl; }

};

// Specialization #1

template <typename T>

class A<T, T>

{

public:

A() { cout << "constructor of template A<T, T>;" << endl; }

};

// Specialization #2

template <typename T>

class A<T, int>

{

public:

A() { cout << "constructor of template A<T, int>;" << endl; }

};

// Specialization #3

template <typename T1, typename T2>

class A<T1\*, T2\*>

{

public:

A() { cout << "constructor of template A<T1\*, T2\*>;" << endl; }

};

int main()

{

A<int> a0; // Template

A<int, float> a1; // Template

A<float, float> a2; // Specialization #1

A<float, int> a3; // Specialization #2

A<int\*, float\*> a4; // Specialization #3

// A<int, int> a5; // Error!!!

// A<int\*, int\*> a6; // Error!!!

}

Пример 08.18. Устранение неоднозначности зависимых имен.

# include <iostream>

template <typename T>

struct S

{

struct Subtype {};

template <typename U>

void f() {}

};

template <typename T>

void g()

{

S<T> s;

s.template f<T>();

}

template <typename T>

void g(const T& t)

{

// T::Subtype\* p; // Error! Идентификатор p не найден

typename T::Subtype\* p;

}

int main()

{

g<int>();

g(S<int>{});

}

Пример 08.19. Шаблон функции с переменным числом параметров.

# include <iostream>

using namespace std;

template <typename Type>

Type sum(Type value)

{

return value;

}

template <typename Type, typename... Args>

Type sum(Type value, Args... params)

{

return value + sum(params...);

}

int main()

{

cout << sum(1, 2, 3, 4, 5) << endl;

}

Пример 08.20. Шаблонный класс Creator с переменным числом параметров (вызов конструктора).

# include <iostream>

using namespace std;

class A

{

public:

A(int k, double d)

{

cout << "Calling the constructor of class A" << endl;

}

};

class Creator

{

public:

template<typename Type, typename... Args>

static Type\* create(Args&&... params)

{

return new Type(forward<Args>(params)...);

}

};

void main()

{

double d = 2.;

A\* p = Creator::create<A>(1, d);

delete p;

}

Пример 08.21. Применение бинарного оператора ко всем аргументам пакета параметров.

# include <iostream>

using namespace std;

# define Prim\_3

# ifdef Prim\_1

template <typename... Ts>

void ignore(Ts...) {}

template <typename T, typename... Ts>

auto sum(T value, Ts... params)

{

auto result = value;

ignore(result += params...);

return result;

}

# elif defined(Prim\_2)

template <typename... Ts>

auto sum(Ts... params)

{

// return (... + params); // (..(p1 + p2) + p3) + ..)

return (params + ...); // (p1 + (p2 + (p3 + ..)..)

}

# elif defined(Prim\_3)

template <typename T, typename... Ts>

auto sum(T v1, Ts... params)

{

// return (v1 + ... + params); // (..(v1 + p1) + p2) + ..)

return (params + ... + v1); // (v1 + (p1 + (p2 + ..)..)

}

# endif Prim\_3

void main()

{

auto s = sum(1, 2, 3, 4);

cout << s << endl;

}

Пример 08.22. Использование вариативных выражений.

# include <iostream>

using namespace std;

template<typename... Ts>

auto sum(Ts... params)

{

return (params + ...);

}

template<typename... Ts>

auto length(Ts... params)

{

return sqrt(sum(params \* params...));

}

int main()

{

cout << length(1., 2., 3., 4., 5.) << endl;

}

Пример 08.23. Использование sizeof.

# include <iostream>

using namespace std;

template <class... Ts>

pair<size\_t, common\_type\_t<Ts...>> sum(Ts... params)

{

return { sizeof...(Ts), (params + ...) };

}

int main()

{

auto [iNumbers, iSum] { sum(1, 2, 3, 4, 5) };

cout << iNumbers << ' ' << iSum << endl;

}

Пример 08.24. Шаблон класса с переменным числом параметров значений.

# include <iostream>

using namespace std;

template <size\_t...>

struct Sum {};

template <>

struct Sum<>

{

enum { value = 0 };

};

template <size\_t val, size\_t... args>

struct Sum<val, args...>

{

enum { value = val + Sum<args...>::value };

};

int main()

{

cout << Sum<1, 2, 3, 4>::value << endl;

}

Пример 08.25. Шаблоны с переменным числом параметров значений.

# include <iostream>

using namespace std;

# define PRIM\_1

# ifdef PRIM\_1

template <size\_t...>

constexpr size\_t sum = 0;

template <size\_t first, size\_t... other>

constexpr size\_t sum<first, other...> = first + sum<other...>;

# elif defined(PRIM\_2)

template <size\_t... Nms>

size\_t sum()

{

auto list = { Nms... };

size\_t sm = 0;

for (auto elem : list)

sm += elem;

return sm;

}

# elif defined(PRIM\_3)

template <typename... Ags>

void stub(Ags...) {}

template <size\_t... Nms>

size\_t sum()

{

size\_t sm = 0;

stub(sm += Nms...);

return sm;

}

# elif defined(PRIM\_4)

template <size\_t... Nms>

size\_t sum()

{

return (Nms + ...);

}

# endif

int main()

{

# ifdef PRIM\_1

cout << sum<1, 2, 3, 4, 5> << endl;

# else

cout << sum<1, 2, 3, 4, 5>() << endl;

# endif

}

Пример 08.26. Использование вариативных выражений для потоков вывода.

# include <iostream>

using namespace std;

template <typename T>

class AddSpace

{

private:

const T& ref;

public:

AddSpace(const T& r) : ref(r) {}

friend ostream& operator <<(ostream& os, AddSpace as)

{

return os << ' ' << as.ref;

}

};

template <typename... Ts>

ostream& print(ostream& os, Ts&&... args)

{

return (os << ... << AddSpace(forward<Ts>(args)));

}

int main()

{

print(cout, 1, 2, 3, 4, 5) << endl;

}

Пример 08.27. Шаблон класса с переменным числом параметров. Рекурсивная реализация кортежа.

# include <iostream>

using namespace std;

template <typename... Types>

class Tuple;

template <typename Head, typename... Tail>

class Tuple<Head, Tail...>

{

private:

Head value;

Tuple<Tail...> tail;

public:

Tuple() = default;

Tuple(const Head& v, const Tuple<Tail...>& t) : value(v), tail(t) {}

Tuple(const Head& v, const Tail&... tail) : value(v), tail(tail...) {}

Head& getHead() { return value; }

const Head& getHead() const { return value; }

Tuple<Tail...>& getTail() { return tail; }

const Tuple<Tail...>& getTail() const { return tail; }

};

template <>

class Tuple<>

{

};

template <size\_t N>

struct Get

{

template <typename Head, typename... Tail>

static auto apply(const Tuple<Head, Tail...>& t)

{

return Get<N - 1>::apply(t.getTail());

}

};

template <>

struct Get<0>

{

template <typename Head, typename... Tail>

static const Head& apply(const Tuple<Head, Tail...>& t)

{

return t.getHead();

}

};

template <size\_t N, typename... Types>

auto get(const Tuple<Types...>& t)

{

return Get<N>::apply(t);

}

size\_t count(const Tuple<>&)

{

return 0;

}

template <typename Head, typename... Tail>

size\_t count(const Tuple<Head, Tail...>& t)

{

return 1 + count(t.getTail());

}

ostream& writeTuple(ostream& os, const Tuple<>&)

{

return os;

}

template <typename Head, typename... Tail>

ostream& writeTuple(ostream& os, const Tuple<Head, Tail...>& t)

{

os << t.getHead() << " ";

return writeTuple(os, t.getTail());

}

template <typename... Types>

ostream& operator<<(ostream& os, const Tuple<Types...>& t)

{

return writeTuple(os, t);

}

int main()

{

Tuple<const char\*, double, int, char> obj("Pi: ", 3.14, 15, '!');

cout << get<0>(obj) << get<1>(obj) << get<2>(obj) << get<3>(obj) << endl;

cout << obj << endl;

cout << "Count = " << count(obj) << endl;

}

Пример 08.28. Использование указателя this.

# include <iostream>

using namespace std;

template<typename Т>

class Base

{

public:

void f() { cout << "method f is called" << endl; }

};

template<typename T>

class Derived : public Base<T>

{

public:

void func()

{

// f(); // идентификатор f не найден

this->f();

}

};

int main()

{

Derived<int> obj{};

obj.func();

}