Пример 09.01. 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 q);

template <typename Type>

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

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

template <typename Type>

void sort(Type\* arr, int q, 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 q)

{

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

cin >> arr[i];

return arr;

}

template <typename Type>

void outputArray(Type\* arr, int q)

{

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

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

cout << endl;

}

template <typename Type>

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

{

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

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

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

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

}

Пример 09.11. Правило вызова функций.

# include <iostream>

template <typename Type>

void swap(Type& val1, Type& val2)

{

Type temp = val1;

val1 = val2;

val2 = 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;

}

void main()

{

const int N = 2;

int a1[N];

float a2[N];

double a3[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&)

}

Пример 09.12. 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 09.02. 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<Type, N>& 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();

const Type\* iter = lt.begin();

int i;

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

arr[i] = \*iter;

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<Type, N>& a) const

{

if (this == &a) return true;

bool Key = true;

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

Key = this->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;

return 0;

}

Пример 09.14. 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 09.15. 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 this->elem + d;

}

int main()

{

A<int> obj(1);

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

}

Пример 08.16. Использование decltype(auto) 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>

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

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

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

template<typename Type>

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

};

template<typename T>

template<typename U>

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

{

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

}

template<typename Type>

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

{

return os << "( " << com.real << ", " << com.imag << " )" << endl;

}

int main()

{

Complex<double> c1(1.1, 2.2);

Complex<int> c2(1, 2);

cout << c2 + c1 << endl;

}

Пример 09.03. 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();

return 0;

}

Пример 09.04. 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 09.05. 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... args)

{

return value + sum(args...);

}

int main()

{

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

return 0;

}

Пример 09.08. Шаблон функции с переменным числом параметров.

# include <iostream>

using namespace std;

template<typename T, typename U>

bool comparer(T&& var1, U&& var2) { return var1 == var2; }

template<typename T, typename U, typename... Args>

bool comparer(T&& var1, U&& var2, Args&&... args)

{

return var1 == var2 && comparer(var2, args...);

}

class Double

{

double elem;

public:

Double(const Double& e) : elem(e.elem)

{

cout << "Calling the copy constructor" << endl;

}

Double(Double&& e) : elem(e.elem)

{

cout << "Calling the transfer constructor" << endl;

}

Double(double e) : elem(e) {}

Double operator -(const Double& e) const;

friend bool operator ==(const Double& e1, const Double& e2);

};

Double Double::operator -(const Double& e) const

{

Double d{ this->elem - e.elem };

return d;

}

bool operator ==(const Double& e1, const Double& e2)

{

return e1.elem == e2.elem;

}

void main()

{

Double e1 = 1., e2 = 2.;

cout << comparer(1, 1U, 1., 1.f, e1, e2 - e1) << endl;

}

Пример 09.06. Шаблон 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;

return 0;

}

Пример 09.10. Шаблон с переменным числом параметров значений.

# include <iostream>

using namespace std;

template <size\_t...>

constexpr size\_t sum = 0;

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

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

int main()

{

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

}

Пример 09.09. Шаблон функции с переменным числом параметров (вызов конструктора).

# include <iostream>

using namespace std;

class A

{

int key;

double data;

public:

A(int k, double d) : key(k), data(d)

{

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

}

};

/\*

template<class Type>

struct myRemove\_reference {

using type = Type;

};

template<class Type>

Type&& myForward(typename myRemove\_reference<Type>::type& t) noexcept {

return static\_cast<Type&&>(t);

}

\*/

class Creator

{

public:

template<typename Type, typename ...Args>

static Type\* create(Args...args)

{

return new Type(args...);

}

/\*

template<typename Type, typename ...Args>

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

{

return new Type(myForward<Args>(args)...);

}

\*/

/\*

template<typename Type, typename ...Args>

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

{

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

}

\*/

};

void main()

{

double d = 2.;

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

delete p;

}

Пример 09.07. 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;

}