Пример 07.01. Обработка исключительных ситуаций.

# include <iostream>

# include <exception>

using namespace std;

class ExceptionArray : public std::exception

{

protected:

static const size\_t sizebuff = 128;

char errormsg[sizebuff]{};

public:

ExceptionArray() noexcept = default;

ExceptionArray(const char\* msg) noexcept

{

strcpy\_s(errormsg, sizebuff, msg);

}

~ExceptionArray() override {}

const char\* what() const noexcept override { return errormsg; }

};

class ErrorIndex : public ExceptionArray

{

private:

const char\* errIndexMsg = "Error Index";

int ind;

public:

ErrorIndex(const char\* msg, int index) noexcept : ind(index)

{

sprintf\_s(errormsg, sizebuff, "%s %s: %4d!", msg, errIndexMsg, ind);

}

~ErrorIndex() override {}

const char\* what() const noexcept override { return errormsg; }

};

void main()

{

try

{

throw(ErrorIndex("Index!!", -1));

}

catch (const ExceptionArray& error)

{

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

}

catch (std::exception& error)

{

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

}

catch (...)

{

cout << "All errors!" << endl;

}

}

Пример 07.02. “Прокидывание” исключения.

# include <iostream>

# include <exception>

using namespace std;

class Exception\_Alloc : public std::exception

{

public:

const char\* what() const noexcept override

{

return "Memory allocation error!";

}

};

class A

{

private:

int\* arr;

public:

A(int size) : arr(new int[size] {}) {}

~A() { delete[] arr; }

};

int main()

{

try

{

try

{

try

{

A\* pobj = new A(-2);

delete pobj;

}

catch (const std::bad\_alloc& err)

{

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

throw Exception\_Alloc();

}

}

catch (const Exception\_Alloc& err)

{

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

throw;

}

}

/\*

catch (const Exception\_Alloc& err)

{

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

}

\*/

catch (...)

{

cout << "All errors!" << endl;

}

}

Пример 07.03. Try и Catch блоки уровня методов.

# include <iostream>

# include <exception>

using namespace std;

class A

{

public:

void f(int v);

};

void A::f(int v) try

{

if (v < 0) throw std::runtime\_error("error in method f!");

}

catch (const std::runtime\_error& err)

{

cout << err.what() << " v = " << v << endl;

}

int main()

{

A obj;

obj.f(-1);

}

Пример 07.04. Блок try для раздела инициализации конструктора.

# include <iostream>

# include <exception>

using namespace std;

class ErrorArrayAlloc : public std::exception

{

public:

const char\* what() const noexcept override

{

return "Errors in allocating memory for an Array!";

}

};

class Array

{

private:

double\* mas;

int cnt;

public:

Array(int q);

~Array() { delete[] mas; }

};

Array::Array(int q) try : mas(new double[q]), cnt(q)

{}

catch (const std::bad\_alloc& exc)

{

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

throw ErrorArrayAlloc();

}

void main()

{

try

{

Array a(-1);

}

catch (const ErrorArrayAlloc& err)

{

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

}

catch (const std::bad\_alloc& exc)

{

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

}

}

Пример 07.05. Цикл for с блоком try.

# include <iostream>

# include <exception>

using namespace std;

class ErrorBase : public std::exception

{

public:

const char\* what() const noexcept override

{

return "Error in the Base";

}

};

# pragma region Errors with the array

class ErrorArray : public std::exception

{

public:

const char\* what() const noexcept override

{

return "Error in the Array";

}

};

class ErrorArraySize : public ErrorArray

{

public:

const char\* what() const noexcept override

{

return "Array size error";

}

};

class ErrorArrayIndex : public ErrorArray

{

public:

const char\* what() const noexcept override

{

return "Array index error";

}

};

# pragma endregion

class Base

{

public:

Base(int size)

{

cout << "Contructor Base" << endl;

if (size < 0) throw ErrorBase();

}

~Base()

{

cout << "Destructor Base" << endl;

}

};

class Array : public Base

{

private:

double\* ar;

int count;

public:

Array(int n) try : Base(n), count(n)

{

cout << "Contructor Array" << endl;

if (this->count <= 0) throw ErrorArraySize();

this->ar = new double[this->count];

}

catch (const ErrorBase& err)

{

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

throw ErrorArray();

}

~Array()

{

cout << "Destructor Array" << endl;

delete[] ar;

}

double& operator [](int index)

{

if (index < 0 || index >= this->count) throw ErrorArrayIndex();

return this->ar[index];

}

};

int main()

{

for (int i = -1; i < 3; i++) try

{

cout << i + 1 << endl;

Array ar(i);

ar[i - 2];

}

catch (const ErrorArray& err)

{

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

}

catch (const ErrorBase& err)

{

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

}

}

Пример 07.06. Метод с условным оператором noexcept.

# include <iostream>

# include <exception>

using namespace std;

class A

{

private:

A(int d) // noexcept

{

if (d < 0)

throw std::runtime\_error("Error!");

}

public:

~A() noexcept(false) // деструктор по умолчанию noexcept(true)

{

throw std::runtime\_error("Destructor");

}

static A create(int v);

};

A A::create(int v) noexcept(noexcept(A(v)))

{

return A(v);

}

int main()

{

try

{

A obj = A::create(-5);

}

catch (const std::runtime\_error& err)

{

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

}

}

Пример 07.07. Код небезопасный относительно исключений.

# include <iostream>

# include <exception>

using namespace std;

class A

{

public:

void operator =(const A& obj)

{

throw std::runtime\_error("Copy error!");

}

};

class Array

{

private:

A\* arr;

int count;

public:

explicit Array(int cnt) try : count(cnt), arr(new A[cnt]{})

{}

catch (const std::bad\_alloc& err)

{

throw;

}

explicit Array(const Array& a);

~Array();

};

Array::~Array()

{

cout << "Destructor!" << endl;

delete[] arr;

}

Array::Array(const Array& a) : count(a.count)

{

arr = new A[count]{};

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

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

}

int main()

{

try

{

Array a1(10);

Array a2{ a1 };

}

catch (const std::runtime\_error& err)

{

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

}

catch (const std::bad\_alloc& err)

{

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

}

}

Пример 07.08. Обертывание исключения в exception\_ptr.

# include <iostream>

# include <exception>

using namespace std;

void do\_raise()

{

throw std::runtime\_error("Exception!");

}

exception\_ptr get\_excption()

{

try

{

do\_raise();

}

catch (...)

{

return current\_exception();

}

return nullptr;

}

int main()

{

try

{

exception\_ptr ex = get\_excption();

rethrow\_exception(ex);

}

catch (const std::runtime\_error& err)

{

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

}

}

Пример 07.09. Вызов деструктора в результате прокидывания исключения.

# include <iostream>

# include <exception>

using namespace std;

class A

{

private:

int count = std::uncaught\_exceptions();

public:

A() = default;

~A()

{

if (count != std::uncaught\_exceptions())

{

cout << "Exception -> Destructor!" << endl;

}

else

{

cout << "Destructor!" << endl;

}

}

void f()

{

throw std::runtime\_error("Exception in method f!");

}

};

int main()

{

try

{

A obj;

obj.f();

}

catch (const std::runtime\_error& err)

{

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

}

}

Пример 07.11. Использование оператора ->\*.

# include <iostream>

using namespace std;

class Callee;

class Caller

{

using FnPtr = int (Callee::\*)(int);

private:

Callee\* pobj;

FnPtr ptr;

public:

Caller(Callee\* p, FnPtr pf) : pobj(p), ptr(pf) {}

int call(int d) { return (pobj->\*ptr)(d); }

};

class Callee

{

private:

int index;

public:

Callee(int i = 0) : index(i) {}

int inc(int d) { return index += d; }

int dec(int d) { return index -= d; }

};

void main()

{

Callee obj;

Caller cl1(&obj, &Callee::inc);

Caller cl2(&obj, &Callee::dec);

cout << " 1: " << cl1.call(3) << "; 2: " << cl2.call(5) << endl;

}

Пример 07.12. Перегрузка бинарных и унарных операторов.

# include <iostream>

using namespace std;

class Complex

{

private:

double re, im;

public:

Complex(double r = 0., double i = 0.) : re(r), im(i) {}

Complex operator-() const { return Complex(-re, -im); }

Complex operator-(const Complex& c) const { return Complex(re + c.re, im + c.im); }

friend Complex operator+(const Complex& c1, const Complex& c2);

friend ostream& operator<<(ostream& os, const Complex& c);

};

Complex operator+(const Complex& c1, const Complex& c2)

{

return Complex(c1.re + c2.re, c1.im + c2.im);

}

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

{

return os << c.re << " + " << c.im << "i";

}

void main()

{

Complex c1(1., 1.), c2(1., 2.), c3(2., 1.);

Complex c4 = c1 + c2;

cout << c4 << endl;

Complex c5 = 5 + c3;

cout << c5 << endl;

// Complex c6 = 6 - c3; Error!!!

Complex c7 = -c1;

cout << c7 << endl;

}

Пример 07.13. “Умные” указатели. Перегрузка операторов -> и \*.

# include <iostream>

using namespace std;

class A

{

public:

void f() const { cout << "Executing f from A;" << endl; }

};

class B

{

private:

A\* pobj;

public:

B(A\* p) : pobj(p) {}

A\* operator->() noexcept { return pobj; }

const A\* operator->() const noexcept { return pobj; }

A& operator\*() noexcept { return \*pobj; }

const A& operator\*() const noexcept { return \*pobj; }

};

void main()

{

A a;

B b1(&a);

b1->f();

const B b2(&a);

(\*b2).f();

}

Пример 07.14. Особенности перегрузки оператора ->.

# include <iostream>

using namespace std;

class A

{

public:

void f() { cout << "Executing f from A;" << endl; }

};

class B

{

private:

A\* pobj;

public:

explicit B(A\* p) : pobj(p) {}

A\* operator->() { cout << "B -> "; return pobj; }

};

class C

{

private:

B& alias;

public:

C(B& b) : alias(b) {}

B& operator->() { cout << "C -> "; return alias; }

};

void main()

{

A a;

B b(&a);

C c(b);

c->f();

}

Пример 07.15. Использование виртуальных операторов -> и \*. Ковариантность.

# include <iostream>

using namespace std;

class A

{

public:

void g() { cout << "A::g" << endl; }

};

class B : public A

{

public:

void g() { cout << "B::g" << endl; }

};

class Base

{

public:

virtual ~Base() = default;

virtual A\* operator ->() = 0;

virtual A& operator \*() = 0;

};

class C : public Base

{

private:

A\* ptr = new A;

public:

~C() override { delete ptr; }

A\* operator ->() override { return ptr; }

A& operator \*() override { return \*ptr; }

};

class D : public Base

{

private:

B\* ptr = new B;

public:

~D() override { delete ptr; }

B\* operator ->() override { return ptr; }

B& operator \*() override { return \*ptr; }

};

int main()

{

D obj;

obj->g();

(\*obj).g();

Base& alias = obj;

alias->g();

(\*alias).g();

}

Пример 07.16. Перегрузка оператора ->\*. Функтор.

# include <iostream>

using namespace std;

class Callee

{

private:

int index;

public:

Callee(int i = 0) : index(i) {}

int inc(int d) { return index += d; }

};

class Caller

{

public:

using FnPtr = int (Callee::\*)(int);

private:

Callee\* pobj;

FnPtr ptr;

public:

Caller(Callee\* p, FnPtr pf) : pobj(p), ptr(pf) {}

int operator ()(int d) { return (pobj->\*ptr)(d); }

};

class Pointer

{

private:

Callee\* pce;

public:

Pointer(int i) { pce = new Callee(i); }

~Pointer() { delete pce; }

Caller operator->\*(Caller::FnPtr pf) { return Caller(pce, pf); }

};

void main()

{

Caller::FnPtr pn = &Callee::inc;

Pointer pt(1);

cout << "Result: " << (pt->\*pn)(2) << endl;

}

Пример 07.26. Этот безумный С++ и оператор ->\*.

# include <iostream>

using namespace std;

auto operator ->\*(pair<int, int>& pr, bool key) -> decltype(key ? pr.first : pr.second)

{

return key ? pr.first : pr.second;

}

auto main() -> int

{

pair t{ 1, 2 };

t->\*true += 2;

auto [f, s]{ t };

cout << "pair{" << f << ", " << s << "}" << endl;

}

Пример 07.17. Перегрузка операторов [], =, ++ и приведения типа.

# include <iostream>

# include <exception>

# include <stdexcept>

using namespace std;

class Index

{

private:

int ind;

public:

Index(int i = 0) : ind(i) {}

Index& operator++() // ++obj

{

++ind;

return \*this;

}

Index operator++(int) // obj++

{

Index it(\*this);

++ind;

return it;

}

operator int() const { return ind; }

};

class Array

{

public:

explicit Array(int n = 0) : cnt(n)

{

mas = cnt > 0 ? new double[cnt] : ((cnt = 0), nullptr);

}

explicit Array(const Array& arr) { copy(arr); }

Array(Array&& arr) noexcept { move(arr); }

~Array() { delete[]mas; }

Array& operator =(const Array& arr);

Array& operator =(Array&& arr) noexcept;

double& operator [](const Index& index);

const double& operator [](const Index& index) const;

int count() const { return cnt; }

private:

double\* mas;

int cnt;

void copy(const Array& arr);

void move(Array& arr) noexcept;

};

Array& Array::operator =(const Array& arr)

{

if (this == &arr) return \*this;

delete []mas;

copy(arr);

return \*this;

}

Array& Array::operator =(Array&& arr) noexcept

{

delete[]mas;

move(arr);

return \*this;

}

double& Array::operator[](const Index& index)

{

if (index < 0 || index >= cnt) throw std::out\_of\_range("Error: class Array operator [];");

return mas[index];

}

const double& Array::operator[](const Index& index) const

{

if (index < 0 || index >= cnt) throw std::out\_of\_range("Error: class Array operator [];");

return mas[index];

}

void Array::copy(const Array& arr)

{

cnt = arr.cnt;

mas = new double[cnt];

memcpy(mas, arr.mas, cnt \* sizeof(double));

}

void Array::move(Array& arr) noexcept

{

cnt = arr.cnt;

mas = arr.mas;

arr.mas = nullptr;

}

Array operator \*(const Array& arr, double d)

{

Array a(arr.count());

for (Index i; i < arr.count(); i++)

a[i] = d \* arr[i];

return a;

}

Array operator \*(double d, const Array& arr) { return arr \* d; }

Array operator +(const Array& arr1, const Array& arr2)

{

if (arr1.count() != arr2.count()) throw std::length\_error("Error: operator +;");

Array a(arr1.count());

for (Index i; i < arr1.count(); i++)

a[i] = arr1[i] + arr2[i];

return a;

}

istream& operator >>(istream& is, Array& arr)

{

for (Index i; i < arr.count(); i++)

cin >> arr[i];

return is;

}

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

{

for (Index i; i < arr.count(); i++)

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

return os;

}

void main()

{

try

{

const int N = 3;

Array a1(N), a2;

cout << "Input of massive (size = " << a1.count() << "): ";

cin >> a1;

// a2 = a1 + 5; Error!!!

a2 = 2 \* a1;

cout << "Result: " << a2 << endl;

}

catch (const std::exception& exc)

{

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

}

}

Пример 07.18. Перегрузка оператора (). Функтор.

# include <iostream>

using namespace std;

class A

{

public:

int operator ()() const { return 0; }

int operator ()(int i) const { return i; }

int operator ()(int i, int j) const { return i + j; }

};

void main()

{

A obj;

cout << obj() << ", " << obj(1) << ", " << obj(1, 2) << endl;

}

Пример 07.19. Оператор new для массива.

# include <iostream>

using namespace std;

class Complex

{

double re, im;

public:

Complex(double r = 0., double i = 0.) : re(r), im(i) {}

double getR() const { return re; }

double getI() const { return im; }

};

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

{

return os << " ( " << c.getR() << ", " << c.getI() << " )";

}

int main()

{

const int count = 10;

Complex\* arr = new Complex[count]{ 1., { 2., 3. }, Complex(4., 5.), 6., 7. };

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

cout << arr[i];

cout << endl;

delete[] arr;

}

Пример 07.20. Перегрузка операторов new, delete.

# include <iostream>

using namespace std;

class A

{

public:

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

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

void\* operator new(size\_t size)

{

cout << "new A" << endl;

return ::operator new(size);

}

void operator delete(void\* ptr)

{

cout << "delete A" << endl;

::operator delete(ptr);

}

void\* operator new[](std::size\_t size)

{

cout << "new[] A" << endl;

return ::operator new[](size);

}

void operator delete[](void\* ptr)

{

cout << "delete[] A" << endl;

::operator delete[](ptr);

}

};

void main()

{

A\* pa = new A;

delete pa;

pa = new A[2];

delete[] pa;

}

Пример 07.21. Перегрузка операторов на примере класс Array.

# include <iostream>

# include <initializer\_list>

# include <exception>

# include <stdexcept>

using namespace std;

class Array final

{

public:

explicit Array(int n = 0, double\* a = nullptr);

explicit Array(const Array& arr) { copy(arr.mas, arr.cnt); }

Array(Array&& arr) noexcept { move(arr); }

Array(initializer\_list<double> list) { copy(list); }

~Array() { delete[]mas; }

Array& operator =(const Array& arr);

Array& operator =(Array&& arr) noexcept;

Array& operator =(initializer\_list<double> list);

double& operator [](int index);

const double& operator [](int index) const;

explicit operator int() const { return cnt; }

int count() const { return cnt; }

Array& operator /=(double d);

Array operator /(double d) const;

Array& operator \*=(double d);

Array operator \*(double d) const;

Array operator -() const;

Array& operator -=(const Array& arr);

Array& operator -=(initializer\_list<double> list);

Array operator -(const Array& arr) const;

private:

double\* mas;

int cnt;

void copy(const double\* a, int n);

void copy(initializer\_list<double> list);

void move(Array& arr) noexcept;

};

Array operator\*(double d, const Array& arr);

# pragma region Methods Array

Array::Array(int n, double\* a)

{

if (n <= 0)

{

cnt = 0; mas = nullptr;

}

else

{

copy(a, n);

}

}

Array& Array::operator =(const Array& arr)

{

if (this == &arr) return \*this;

delete[] mas;

copy(arr.mas, arr.cnt);

return \*this;

}

Array& Array::operator =(Array&& arr) noexcept

{

delete[] mas;

move(arr);

return \*this;

}

Array& Array::operator =(initializer\_list<double> list)

{

delete[] mas;

copy(list);

return \*this;

}

double& Array::operator [](int index)

{

if (index < 0 || index >= cnt) throw std::out\_of\_range("Error: class Array operator [];");

return mas[index];

}

const double& Array::operator [](int index) const

{

if (index < 0 || index >= cnt) throw std::out\_of\_range("Error: class Array operator [];");

return mas[index];

}

void Array::copy(const double\* a, int n)

{

cnt = n;

mas = new double[cnt];

if (a)

{

memcpy(mas, a, cnt \* sizeof(double));

}

}

void Array::copy(initializer\_list<double> list)

{

cnt = list.size();

mas = new double[cnt];

for (int i = 0; auto elem : list)

mas[i++] = elem;

}

void Array::move(Array& arr) noexcept

{

cnt = arr.cnt;

mas = arr.mas;

arr.mas = nullptr;

}

Array& Array::operator /=(double d)

{

if (d == 0.) throw std::invalid\_argument("Error: divide by zero;");

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

mas[i] /= d;

return \*this;

}

Array Array::operator /(double d) const

{

Array a(\*this);

a /= d;

return a;

}

Array& Array::operator \*=(double d)

{

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

mas[i] \*= d;

return \*this;

}

Array Array::operator \*(double d) const

{

Array a(\*this);

a \*= d;

return a;

}

Array Array::operator -() const

{

return -1. \* (\*this);

}

Array& Array::operator -=(const Array& arr)

{

if (cnt != arr.cnt) throw std::length\_error("Error: operator -;");

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

mas[i] -= arr[i];

return \*this;

}

Array& Array::operator -=(initializer\_list<double> list)

{

if (cnt != list.size()) throw std::length\_error("Error: operator -;");

for (int i = 0; auto elem : list)

mas[i++] -= elem;

return \*this;

}

Array Array::operator-(const Array& arr) const

{

Array a(\*this);

a -= arr;

return a;

}

#pragma endregion

Array operator\*(double d, const Array& arr) { return arr \* d; }

istream& operator >>(istream& is, Array& arr)

{

for (int i = 0; i < arr.count(); i++)

is >> arr[i];

return is;

}

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

{

for (int i = 0; i < arr.count(); i++)

os << " " << arr[i];

return os;

}

void main()

{

try

{

const int N = 3;

Array a1(N), a2, a4{ 2., 4., 6. };

cout << "Input of massive (size = " << a1.count() << "): ";

cin >> a1;

cout << "Result a1: " << a1 << endl;

a2 = 2. \* a1;

cout << "Result a2: " << a2 << endl;

Array a3 = -a1;

cout << "Result a3: " << a3 << endl;

a4 -= {3., 2., 1.};

cout << "Result a4: " << a4 << endl;

Array a5 = a2 - a3;

cout << "Result a5: " << a5 << endl;

}

catch (const exception& exc)

{

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

}

}

Пример 07.22. Оператор приведения типа с автоматическим выведением типа.

# include <iostream>

class A

{

private:

int val;

public:

A(int i) : val(i) {}

operator auto() const& { return val; }

operator auto()&& { return val; }

operator auto\* () const { return &val; }

};

int main()

{

A obj{ 10 };

int v1 = obj; // operator auto() const&

double v2 = obj; // operator auto() const&

const double& al = obj; // operator auto() const&

int v3 = std::move(obj); // operator auto()&&

const int\* p = obj; // operator auto\*() const

}

Пример 07.23. Оператор “space ship”.

# define \_CRT\_SECURE\_NO\_WARNINGS

# include <iostream>

# include <compare>

# include <string.h>

using namespace std;

class MyInt

{

public:

constexpr MyInt(int val) : value{ val } { }

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

private:

int value;

};

class MyDouble

{

public:

constexpr MyDouble(double val) : value{ val } { }

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

private:

double value;

};

class MyString

{

public:

constexpr MyString(const char\* val) : value{ val } { }

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

private:

const char\* value;

};

int main()

{

MyInt i1{ 1 }, i2{ 2 };

cout << (i1 < i2) << endl;

MyDouble d1{ -0. }, d2{ 0. };

cout << (d1 != d2) << (1. < d2) << (d1 < 2.) << endl;

char st[5];

strcpy(st, "Ok!!");

MyString s1{ "Ok!" }, s2{ st };

cout << (s1 < s2) << ("Ok!!" == s2) << endl; // сравнение адресов

}

Пример 07.24. Варианты перегрузки оператора “space ship”.

# include <iostream>

# include <compare>

using namespace std;

class MyInt

{

private:

int value;

public:

MyInt(int val = 0) : value(val) {}

//strong\_ordering operator <=>(const MyInt& rhs) const

//{

// return value <=> rhs.value;

//}

//strong\_ordering operator <=>(const MyInt& rhs) const

//{

// return value == rhs.value ? strong\_ordering::equal :

// value < rhs.value ? strong\_ordering::less :

// strong\_ordering::greater;

//}

//weak\_ordering operator <=>(const MyInt& rhs) const

//{

// return value == rhs.value ? weak\_ordering::equivalent :

// value < rhs.value ? weak\_ordering::less :

// weak\_ordering::greater;

//}

partial\_ordering operator <=>(const MyInt& rhs) const

{

return value == rhs.value ? partial\_ordering::equivalent :

value < rhs.value ? partial\_ordering::less :

value > rhs.value ? partial\_ordering::greater :

partial\_ordering::unordered;

}

bool operator ==(const MyInt&) const = default;

};

int main()

{

MyInt a{ 1 }, b{ 2 }, c{ 3 }, d{ 1 };

cout << "a < b: " << (a < b) << ", c > b: " << (c >= b) << endl;

cout << "a < b: " << (a < b) << ", c > b: " << (c > b) << ", a != b: " << (a != b) << endl;

cout << "a < 5: " << (a < 5) << ", 1 < c: " << (1 < c) << endl;

}

Пример 07.25. Определение литеральных операторов.

# include <iostream>

# include <assert.h>

using namespace std;

unsigned long long operator "" \_b(const char\* str)

{

size\_t size = strlen(str);

unsigned long long result = 0;

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

{

assert(str[i] == '1' || str[i] == '0');

(result <<= 1) |= str[i] - '0';

}

return result;

}

double operator"" \_kg(long double val)

{

return val;

}

int main()

{

cout << 101100\_b << endl;

cout << 76.3\_kg << endl;

}