Пример 07.01. Обработка исключительных ситуаций.

# include <iostream>

# include <exception>

using namespace std;

class ExceptionArray : public std::exception

{

protected:

char\* errormsg;

public:

ExceptionArray(const char\* msg)

{

int Len = strlen(msg) + 1;

this->errormsg = new char[Len];

strcpy\_s(this->errormsg, Len, msg);

}

virtual ~ExceptionArray() { delete[] errormsg; }

virtual const char\* what() const noexcept override { return this->errormsg; }

};

class ErrorIndex : public ExceptionArray

{

private:

const char\* errIndexMsg = "Error Index";

int ind;

public:

ErrorIndex(const char\* msg, int index) : ExceptionArray(msg), ind(index) {}

virtual ~ErrorIndex() {}

virtual const char\* what() const noexcept override

{

int Len = strlen(errormsg) + strlen(errIndexMsg) + 8;

char\* buff = new char[Len + 1];

sprintf\_s(buff, Len, "%s %s: %4d", errormsg, errIndexMsg, ind);

char\* temp = errormsg;

delete[]temp;

const\_cast<ErrorIndex\*>(this)->errormsg = buff;

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 (...)

{

}

}

Пример 07.02. Блок try для раздела инициализации конструктора.

# include <iostream>

# include <exception>

using namespace std;

class ErrorArrayAlloc : public std::exception

{

public:

virtual 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 07.12. Блок try для раздела инициализации конструктора.

# include <iostream>

# include <exception>

using namespace std;

class ErrorBase : public std::exception

{

public:

virtual const char\* what() const noexcept override

{

return "Error in the Base";

}

};

class Base

{

public:

Base(int size)

{

cout << "Contructor Base" << endl;

if (size < 0) throw ErrorBase();

}

~Base()

{

cout << "Destructor Base" << endl;

}

};

# pragma region Errors with the array

class ErrorArray : public std::exception

{

public:

virtual const char\* what() const noexcept override

{

return "Error in the Array";

}

};

class ErrorArraySize : public ErrorArray

{

public:

virtual const char\* what() const noexcept override

{

return "Array size error";

}

};

class ErrorArrayIndex : public ErrorArray

{

public:

virtual const char\* what() const noexcept override

{

return "Array index error";

}

};

# pragma endregion

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 07.03. 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 07.04. 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 07.05. 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->() { return pobj; }

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

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

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

};

void main()

{

A a;

B b1(&a);

b1->f();

const B b2(&a);

(\*b2).f();

}

Пример 07.06. 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 07.07. 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 07.08. 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

{

private:

double\* mas;

int cnt;

void copy(const Array& arr);

void move(Array& arr);

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) { move(arr); }

~Array() { delete[]mas; }

Array& operator=(const Array& arr);

Array& operator=(Array&& arr);

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

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

int count() const { return cnt; }

};

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

{

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

delete[]mas;

copy(arr);

return \*this;

}

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

{

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)

{

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: ";

cin >> a1;

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

a2 = 2 \* a1;

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

}

catch (const std::exception& exc)

{

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

}

}

Пример 07.10. Перегрузка оператора ().

# 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 07.11. 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;

}

Пример 07.09. 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 07.13. Перегрузка операторов на примере класс Array.

# include <iostream>

# include <initializer\_list>

# include <exception>

# include <stdexcept>

# include <cstring>

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;

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;

};

# 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[] this->mas;

copy(arr.mas, arr.cnt);

return \*this;

}

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

{

delete[] this->mas;

move(arr);

return \*this;

}

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

{

delete[] this->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)

{

this->cnt = n;

this->mas = new double[this->cnt];

if (a)

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

}

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

{

this->cnt = list.size();

this->mas = new double[this->cnt];

int i = 0;

for (double elem : list)

this->mas[i++] = elem;

}

void Array::move(Array& arr) noexcept

{

this->cnt = arr.cnt;

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

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

this->mas[i] -= arr[i];

return \*this;

}

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

{

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

int i = 0;

for (double elem : list)

this->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: ";

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;

}

}