Пример 1. Обработка исключительных ситуаций.

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;

}

};

int main()

{

try

{

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

}

catch (ExceptionArray& error)

{

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

}

catch (std::exception& error)

{

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

}

catch (...)

{

}

return 0;

}

Пример 2. Блок try для раздела инициализации конструктора.

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;

}

void main()

{

Array a(-1);

}

Пример 3. Использование оператора ->\*.

class Callee;

class Caller

{

typedef int (Callee::\*FnPtr)(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 4. Перегрузка бинарных и унарных операторов.

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 5. Умные указатели. Перегрузка операторов -> и \*.

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 6. Особенности перегрузки оператора ->.

class A

{

public:

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

};

class B

{

private:

A\* pobj;

public:

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 7. Перегрузка оператора ->\*. Функтор.

class Callee

{

private:

int index;

public:

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

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

};

class Caller

{

public:

typedef int (Callee::\*FnPtr)(int);

private:

Callee\* pobj;

FnPtr ptr;

public:

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

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

};

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 8. Перегрузка операторов [], =, ++ и приведения типа.

# include <iostream>

# include <exception>

# include <stdexcept>

# include <cstring>

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 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 exception& exc)

{

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

}

}

Пример 9. Перегрузка операторов new, delete.

class A

{

// ...

public:

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[1];

delete[] pa;

}