Пример 06.01. Инициализация объектов.

# include <iostream>

# include <initializer\_list>

using namespace std;

class Complex

{

private:

double \_re, \_im;

public:

// explicit

Complex() = default;

// explicit

Complex(double r) : Complex(r, 0.) {}

Complex(int r) : Complex(r, 0.) {}

// explicit

Complex(double r, double i) : \_re(r), \_im(i) {}

// explicit

Complex(const Complex& C) : Complex(C.\_re, C.\_im) {}

Complex(Complex&& C) : Complex(C.\_re, C.\_im) {}

// explicit

Complex(initializer\_list<double> list)

{

for (double elem : list) {}

// for (const double\* p = list.begin(); p != list.end(); p++) {}

}

void set\_Real(int r) { this->\_re = r; }

void set\_Real(double) = delete;

static Complex sum(const Complex& c1, const Complex& c2)

{

Complex ctmp(c1.\_re + c2.\_re, c1.\_im + c2.\_im);

return ctmp;

}

Complex& operator=(const Complex& c)

{

this->\_re = c.\_re; this->\_im = c.\_im;

return \*this;

}

Complex& operator=(Complex&& c)

{

this->\_re = c.\_re; this->\_im = c.\_im;

return \*this;

}

/\*

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

{

return \*this;

}

\*/

};

int main()

{

Complex a1(), // объявление функции

a2(Complex()), // объявление функции

b1, // явный вызов Complex()

b2{}, // явный вызов Complex(). инициализация 0

b3 = {}, // неявный вызов Complex(). инициализация 0

b4((Complex())), // явный вызов Complex(). инициализация 0

b5(Complex{}), // явный вызов Complex(). инициализация 0

b6 = Complex{}, // явный вызов Complex(). инициализация 0

c1\_1(1.5), c1\_2(1), // явный вызов Complex(double), Complex(int)

c2 = Complex(5.5), // явный вызов Complex(double)

c3{ 2. }, // явный вызов Complex(double) | Complex(initializer\_list)

c4 = { 3. }, // неявный вызов Complex(double) | Complex(initializer\_list)

c5 = Complex({ 4. }), // явный вызов Complex(double) | Complex(initializer\_list)

c6 = 4.5, // неявный вызов Complex(double)

d1(1., 2.), // явный вызов Complex(double, double)

d4 = Complex(4., 5.), // явный вызов Complex(double, double)

d2{ 2., 3. }, // Complex(double, double) | Complex(initializer\_list)

d3 = { 3., 4. }, // неявный вызов

// Complex(double, double) | Complex(initializer\_list)

d5 = Complex({ 5., 6. }), // неявный вызов

// Complex(double, double) | Complex(initializer\_list)

e1(c1\_1), // явный вызов Complex(const Complex&)

e2 = Complex(c2), // явный вызов Complex(const Complex&)

e3{ c3 }, // явный вызов Complex(const Complex&)

e4 = { c4 }, // неявный вызов Complex(const Complex&)

e5 = Complex({ c5 }), // неявный вызов Complex(const Complex&)

e6 = c6, // неявный вызов Complex(const Complex&)

f1(Complex::sum(c1\_1, c2)); // вызов Complex(Complex&&)

b1 = {}; // неявный вызов ( Complex() & operator=(Complex&&) )

// | operator=(initializer\_list)

b2 = Complex{}; // явный вызов Complex() & operator=(Complex&&)

c1\_1 = { 1. }; // неявный вызов ( Complex(double) & operator=(Complex&&) )

// | ( Complex(initializer\_list) & operator=(Complex&&) )

// | operator=(initializer\_list)

c4 = 4.; // неявный вызов Complex(double) & operator=(Complex&&)

d1 = { 2., 3. }; // неявный вызов ( Complex(double, double) & operator=(Complex&&) )

// | ( Complex(initializer\_list) & operator=(Complex&&) )

// | operator=(initializer\_list)

e1.set\_Real(1); // '1', 1l, L'1' - Ok! 1ll, 1u, 1lu, 1.f, 1.l, "1" - Error!

return 0;

}

Пример 06.02. Прямая и косвенная базы.

# include <iostream>

using namespace std;

class A

{

public:

A(const char\* s) { cout << "Creature A" << s << ";" << endl; }

};

class B : public A

{

public:

B() : A(" from B") { cout << "Creature B;" << endl; }

};

class C : public B, public A

{

public:

C() : A(" from C") { cout << "Creature C;" << endl; }

};

void main()

{

C obj;

}

Пример 06.03. Виртуальное наследование.

# include <iostream>

using namespace std;

class A

{

public:

A(const char\* s) { cout << "Creature A" << s << ";" << endl; }

};

class B : virtual public A

{

public:

B() : A(" from B") { cout << "Creature B;" << endl; }

};

class C : public B, virtual public A

{

public:

C() : A(" from C") { cout << "Creature C;" << endl; }

};

void main()

{

C obj;

}

Пример 06.04. Виртуальное наследование. Вызов конструкторов.

# include <iostream>

using namespace std;

class A

{

public:

// A() { cout << "Creature A;" << endl; }

A(const char\* s) { cout << "Creature A" << s << ";" << endl; }

};

class B

{

public:

// B() { cout << "Creature B;" << endl; }

B(const char\* s) { cout << "Creature B" << s << ";" << endl; }

};

class C : virtual public A, /\*virtual\*/ public B

{

public:

C(const char\* s) : A(" from C"), B(" from C") { cout << "Creature C" << s << ";" << endl; }

};

class D : virtual public A, /\*virtual\*/ public B

{

public:

D(const char\* s) : A(" from D"), B(" from D") { cout << "Creature D" << s << ";" << endl; }

};

class E : /\*virtual\*/ public C, virtual public D

{

public:

E() : C(" from E"), D(" from E"), A(" from E") { cout << "Creature E;" << endl; }

};

void main()

{

E obj;

}

Пример 06.05. Доминирование.

# include <iostream>

using namespace std;

class A

{

public:

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

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

};

class B : virtual public A

{

public:

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

using A::f;

};

class C : virtual public A {};

class D : virtual public C, virtual public B {};

void main()

{

D obj;

obj.f();

obj.f(1);

}

Пример 06.06. Доминирование и множественное наследование.

# include <iostream>

using namespace std;

class A

{

public:

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

};

class B : virtual public A

{

public:

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

};

class C : public B, virtual public A

{

};

void main()

{

C obj;

obj.f();

}

Пример 06.07. Множественный вызов методов.

# include <iostream>

using namespace std;

class A

{

public:

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

};

class B : virtual public A

{

public:

void f()

{

A::f();

cout << "Executing f from B;" << endl;

}

};

class C : virtual public A

{

public:

void f()

{

A::f();

cout << "Executing f from C;" << endl;

}

};

class D : virtual public C, virtual public B

{

public:

void f()

{

C::f();

B::f();

cout << "Executing f from D;" << endl;

}

};

void main()

{

D obj;

obj.f();

}

Пример 06.08. Решение проблемы множественного вызова методов.

# include <iostream>

using namespace std;

class A

{

protected:

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

public:

void f() { this->\_f(); }

};

class B : virtual public A

{

protected:

void \_f() { cout << "Executing f from B;" << endl; }

public:

void f()

{

A::\_f();

this->\_f();

}

};

class C : virtual public A

{

protected:

void \_f() { cout << "Executing f from C;" << endl; }

public:

void f()

{

A::\_f();

this->\_f();

}

};

class D : virtual public C, virtual public B

{

protected:

void \_f() { cout << "Executing f from D;" << endl; }

public:

void f()

{

A::\_f(); C::\_f(); B::\_f();

this->\_f();

}

};

void main()

{

D obj;

obj.f();

}

Пример 06.09. Неоднозначности при множественном наследовании.

|  |  |
| --- | --- |
| class A  {  public:  int a;  int (\*b)();  int f();  int f(int);  int g();  }; | class B  {  int a;  int b;  public:  int f();  int g;  int h();  int h(int);  }; |
| class C: public A, public B {}; | |

class D

{

public:

static void fun(C& obj)

{

obj.a = 1; // Error!!!

obj.b(); // Error!!!

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

obj.f(1); // Error!!!

obj.g = 1; // Error!!!

obj.h(); obj.h(1); // Ok!

}

};

void main()

{

C obj;

D::fun(obj);

}

Пример 06.10. Подмена методов по одной и ветви.

# include <iostream>

using namespace std;

class A

{

public:

void f1() { cout << "Executing f1 from A;" << endl; }

void f2() { cout << "Executing f2 from A;" << endl; }

};

class B

{

public:

void f1() { cout << "Executing f1 from B;" << endl; }

void f3() { cout << "Executing f3 from B;" << endl; }

};

class C : private A, public B {};

class D

{

public:

void g1(A& obj)

{

obj.f1(); obj.f2();

}

void g2(B& obj)

{

obj.f1(); obj.f3();

}

};

void main()

{

C obj;

D d;

// obj.f1(); Error!!! Множественное определение

// d.g1(obj); Error!!! Нет приведения к базовому классу при наследовании по схеме private

d.g2(obj);

}

Пример 06.11. Объединение интерфейсов.

# include <iostream>

using namespace std;

class A

{

public:

void f1() { cout << "Executing f1 from A;" << endl; }

void f2() { cout << "Executing f2 from A;" << endl; }

};

class B

{

public:

void f1() { cout << "Executing f1 from B;" << endl; }

void f3() { cout << "Executing f3 from B;" << endl; }

};

class C : public A, public B {};

class D

{

public:

void g1(A& obj)

{

obj.f1(); obj.f2();

}

void g2(B& obj)

{

obj.f1(); obj.f3();

}

};

void main()

{

C obj;

D d;

d.g1(obj);

d.g2(obj);

}

Пример 06.12. Виртуальные методы.

# include <iostream>

using namespace std;

class A

{

public:

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

};

class B : public A

{

public:

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

};

class C

{

public:

static void g(A& obj) { obj.f(); }

};

void main()

{

B obj;

C::g(obj);

}

Пример 06.13. Абстрактный класс. Чисто виртуальные методы.

# include <iostream>

using namespace std;

class A // abstract

{

public:

virtual void f() = 0;

};

class B : public A

{

public:

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

};

class C

{

public:

static void g(A& obj) { obj.f(); }

};

void main()

{

B obj;

C::g(obj);

}

Пример 06.14. Чисто виртуальный деструктор.

# include <iostream>

using namespace std;

class A

{

public:

virtual ~A() = 0;

};

A::~A() = default;

class B : public A

{

public:

~B() override { cout << "Class B destructor called;" << endl; }

};

void main()

{

A\* pobj = new B;

delete pobj;

}

Пример 06.15. Вызов виртуальных методов в конструкторах и деструкторах.

# include <iostream>

using namespace std;

class A

{

public:

virtual ~A() { cout << "Class A destructor called;" << endl; }

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

};

class B : public A

{

public:

B() { this->f(); }

~B() override

{

this->f();

cout << "Class B destructor called;" << endl;

}

void g() { this->f(); }

};

class C : public B

{

public:

~C() override { cout << "Class C destructor called;" << endl; }

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

};

void main()

{

C obj;

obj.g();

}

Пример 06.16. Выведение типа возвращаемого значения методом.

# include <iostream>

using namespace std;

class Base

{

public:

~Base() = default;

virtual const int& f() const& = 0;

virtual int f() && = 0;

};

class Derived final : public Base

{

public:

auto f() const& -> const int& final { cout << "Derived::f() const&" << endl; return 0; }

auto f() && -> int override { cout << "Derived::f()&&" << endl; return 0; }

auto g() const { cout << "Derived::g() const" << endl; return 0; }

auto g() { cout << "Derived::g()" << endl; return 0.; }

};

int main()

{

const Derived child1{};

Derived child2;

Base& obj = child2;

decltype(auto) d1 = obj.f();

decltype(auto) d2 = move(obj).f();

auto d3 = child1.g();

auto d4 = child2.g();

}

Пример 06.17. Проблема массива объектов.

# include <iostream>

using namespace std;

class A

{

public:

virtual ~A() = default;

virtual void f() = 0;

};

class B : public A

{

int b;

public:

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

};

class C

{

public:

static A& index(A\* p, int i) { return p[i]; }

};

void main()

{

const int N = 10;

B vect[N];

A& alias = C::index(vect, 5);

alias.f(); // Error!!!

}

Пример 06.18. Проблема передачи объектов в метод по значению.

# include <iostream>

using namespace std;

class A

{

public:

virtual ~A() = default;

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

};

class B : public A

{

public:

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

};

class C

{

public:

static void g(A obj) { obj.f(); }

};

void main()

{

const A& obj = B{};

C::g(obj);

}

Пример 06.19. Множественное наследование и виртуальные методы.

# include <iostream>

using namespace std;

class A

{

public:

virtual ~A() = 0;

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

};

A::~A() {}

class B

{

public:

virtual ~B() = 0;

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

};

B::~B() {}

class C : private A, public B

{

public:

~C() override {}

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

};

class D

{

public:

void g1(A& obj)

{

obj.f();

}

void g2(B& obj)

{

obj.f();

}

};

void main()

{

C obj;

D d;

d.g2(obj);

d.g2(obj);

}

Пример 06.20. Дружба и наследование.

# include <iostream>

using namespace std;

class C; // forward объявление

class A

{

private:

void f1() { cout << "Executing f1;" << endl; }

friend C;

};

class B : public A

{

private:

void f2() { cout << "Executing f2;" << endl; }

};

class C

{

public:

static void g1(A& obj) { obj.f1(); }

static void g2(B& obj)

{

obj.f1();

// obj.f2(); // Error!!! Имеет доступ только к членам A

}

};

class D : public C

{

public:

// static void g2(A& obj) ( obj.f1(); } // Error!!! Дружба не наследуется

};

void main()

{

A aobj;

C::g1(aobj);

B bobj;

C::g1(bobj);

C::g2(bobj);

}

Пример 06.21. Дружба и виртуальные методы.

# include <iostream>

using namespace std;

class C; // forward объявление

class A

{

protected:

virtual ~A() = default;

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

friend C;

};

class B : public A

{

protected:

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

};

class C

{

public:

static void g(A& obj) { obj.f(); }

};

void main()

{

B bobj;

C::g(bobj);

}