Приведение типов

10 июля 2017 г.

Приведение типов

- преобразование данных одного типа в другой

в стиле С

- `int x = 7;`
- `float y = (float) x;`
- `char z = (char) x;`

средствами С++

- используются специальные функции, которые проверяют возможность приведения

Приведение типов в C++

- const_cast (удаляет атрибуты const и volatile)
- static_cast (преобразует неполиморфные типы)
- dynamic_cast (преобразует полиморфные типы)
- reinterpret_cast (повторно интерпретирует разряды)

Синтаксис

```
оператор<итоговый_тип>(выражение);
```

- позволяет изменить тип выражения, если преобразование является допустимым
- большая безопасность, чем при преобразованиях в стиле C

```
const int x = 100;
const int* px = &x;
int* y = const_cast<int*>(px);
char z = static_cast<char>(x);
std::string* str = reinterpret_cast<std::string*>(x);
```

const_cast

```
const_cast<итоговый_тип>(выражение);
```

- снимает действие атрибутов const и volatile
- сам тип (int, char, std::string и т. п.) не меняется
- тип должен быть указателем или ссылкой
- для указателей и ссылок результат будет указывать на исходный объект

```
int x = 100;
const int* px = &x;
int* y = const_cast<int*>(px);
*y = 55;      // x = 55
```

const_cast (пример)

```
class Number {
  int number = 8;            // можно добавить mutable
public:
   void printNumber() const;
};

void Number::printNumber() const {
   (const_cast<Number*>(this))->number--;
   std::cout << number << std::endl;
}

void test() {
   Number obj;
   obj.printNumber();
}
```

static_cast

```
static_cast<итоговый_тип>(выражение);
```

- используются встроенные правила приведения типов или правила, заданные программистом
- не выполняет проверку типа при выполнении
- обычно используют для стандартных числовых преобразований
- может использоваться для приведения указателей на объекты классов

static_cast (пример 1)

```
void f() {
    char ch = 'a';
    int i = 65;
    float f = 2.5;

    std::cout << static_cast<char>(i) << std::endl;      // A

    std::cout << static_cast<double>(f) << std::endl; // 2.5
    std::cout  << sizeof(f) << " " <<                    // 4
        sizeof(static_cast<double>(f)) << std::endl;     // 8

    std::cout << static_cast<int>(ch) << std::endl;      // 97

}
```

static_cast (пример 2)

```
class Base { … };

class Derived : public Base { … };

void f(Base* pbase, Derived* pder) {
        Base* pbase2 = static_cast<Base*>(pder);
        // безопасно

        Derived* pder2 = static_cast<Derived*>(pbase);
        // небезопасно
}
```

dynamic_cast

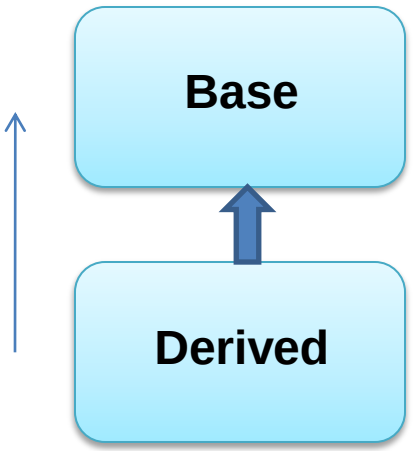
```
dynamic_cast<итоговый_тип>(выражение);
```

- итоговый тип должен быть указателем или ссылкой на тип полиморфного класса
- безопасное приведение по иерархии наследования
- используется RTTI (Runtime Type Information)
- если приведение указателя невозможно, то возвращается NULL; если приведение ссылки невозможно, то вызывается bad_cast

dynamic_cast (пример 1)

- хорошо поддерживаются восходящие преобразования

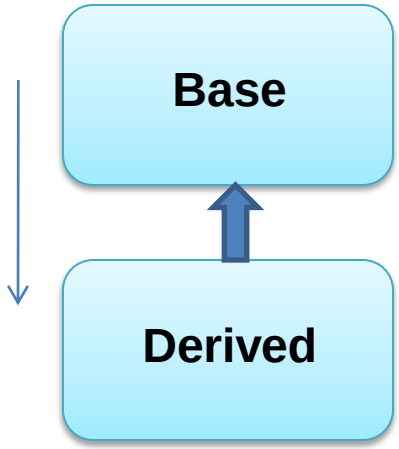
```
class Base {
public:
   virtual void print();
};

class Derived : public Base { … };

void f(Base* pb1) {
   Base* pbase = dynamic_cast<Base*>(pb1);
}

// Base* pb1 = new Base | Base* pb1 = new Derived?
```



dynamic_cast (пример 2)

- поддерживаются допустимые нисходящие преобразования

```
class Base {
public:
   virtual void print();
};

class Derived : public Base { … };

void f(Base* pb1) {
   Derived* pder = dynamic_cast<Derived*>(pb1);
}

// Base* pb1 = new Base | Base* pb1 = new Derived?
                Error          |          OK
```



dynamic_cast (пример 3)

- при нисходящем приведении обязательна проверка на NULL

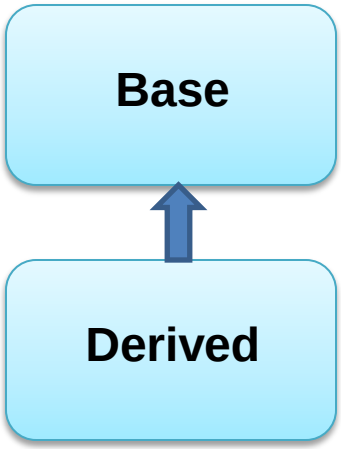
```
class Base {
public:
   virtual void print();
};

class Derived : public Base { … };

void f(Base* pb1) {
      Derived* pder = dynamic_cast<Derived*>(pb1);
      if (pder) {
         pder->print();
      }
}
```

static_cast и dynamic_cast

- могут использоваться для преобразования указателя на базовый класс в указатель на производный класс

static_cast и dynamic_cast (пример)

```
class Base {
public:
   virtual void print(){}
};

class Derived : public Base {};

void f(Base* pbase) {
   Derived* pderived1 = dynamic_cast<Derived*>(pbase);
   Derived* pderived2 = static_cast<Derived*>(pbase);
}

// Base* pbase = new Base | Base* pbase = new Derived ?
```



reinterpret_cast

```
reinterpret_cast<итоговый_тип>(выражение);
```

- самое небезопасное преобразование
- можно привести указатель к указателю, указатель к числу, число к указателю

```
void f() {
    int x = 1000;
    int* px = &x;
    std::cout << reinterpret_cast<int*>(x) << std::endl;
    std::cout << reinterpret_cast<int>(px) << std::endl;
}
```

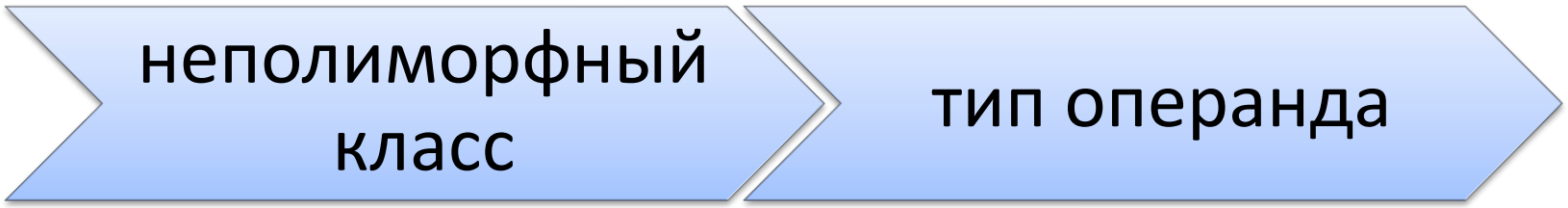
Оператор typeid

- позволяет определить фактический тип выражения
- нужно подключить typeinfo

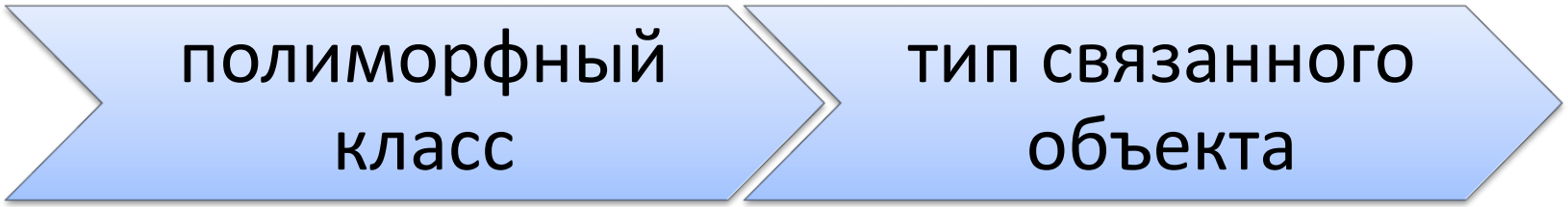
```
#include <typeinfo>

void f() {
    int x = 1000;
    double y = 1.2;
    std::cout << typeid(x).name() << std::endl;  // int | i
    std::cout << typeid(y).name() << std::endl; // double | d
}
```

typeid и наследование

- если операнд имеет тип неполиморфного класса, то typeid возвращает тип операнда



- если операнд имеет тип полиморфного (с виртуальными методами) класса, то typeid возвращает тип связанного с ним объекта



typeid и наследование (пример 1)

```
class Base {
public:
    void print();
};

class Derived : public Base { … };

void f() {
    Base* pb1 = new Base;
    Base* pb2 = new Derived;
    std::cout << typeid(*pb1).name() << std::endl;  // class Base
    std::cout << typeid(*pb2).name() << std::endl;  // class Base
}
```

typeid и наследование (пример 2)

```
class Base {
public:
   virtual void print();
};

class Derived : public Base { … };

void f() {
  Base* pb1 = new Base;
  Base* pb2 = new Derived;
  std::cout << typeid(*pb1).name() << std::endl;  // class Base
  std::cout << typeid(*pb2).name() << std::endl;  // class Derived
}
```

typeid и наследование (пример 3)

```
class Base {
public:
   virtual void print();
};

class Derived : public Base { … };

void f() {
     Base* pb1 = new Base;
     Base* pb2 = new Derived;
     std::cout << typeid(pb1).name() << std::endl;  // class Base *
     std::cout << typeid(pb2).name() << std::endl;  // class Base *
}
```

Какие приведения завершатся неудачно?

Дана иерархия классов, в которой у каждого класса есть конструктор по умолчанию и виртуальный деструктор:

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
class D : public X, public C { ... };
```

1)
```
D *pd = new D;
A *pa = dynamic_cast< A* > ( pd );
```

Какие приведения завершатся неудачно?

Дана иерархия классов, в которой у каждого класса есть конструктор по умолчанию и виртуальный деструктор:

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
class D : public X, public C { ... };
```

2)
```
A *pa = new C;
C *pc = dynamic_cast< C* > ( pa );
```

Какие приведения завершатся неудачно?

Дана иерархия классов, в которой у каждого класса есть конструктор по умолчанию и виртуальный деструктор:

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
class D : public X, public C { ... };
```

3)
```
B *pb = new B;
D *pd = dynamic_cast< D*> ( pb );
```

Вопросы?